UiO : Department of Informatics
University of Oslo

Creating a chunk structure for Norwegian

Fredrik Wilhelmsen

Master's Thesis Spring 2015

Contents

Introduction

This thesis is divided into two distinct parts. The first part, contained in part I, is about building a chunk structure. The goal there is to create rules for using a dependency parse to create chunks for a sentence, then using those rules to create a chunk structure for the sentences in the Norwegian Dependency Treebank (*Norsk dependenstrebank* 2014).

The second part, contained in part II, is creating and evaluating an evolutionary algorithm to predict chunks in a sentence.

Part I

From dependency graph to a chunk structure

About Part I

The most used methods for making a chunked set of sentences, have been to either manually tag a set, or to extract the chunks from an already parsed structure, for example from the Penn Treebank. The goal in this part of the thesis, however, is to use a dependency-parsed corpus, with the included part of speech tags, to make a chunked corpus.

This part documents a method for using the Norwegian Dependency Treebank (*Norsk dependenstrebank* 2014) to create a chunked structure for the sentences in the NDT.

Chapter 1

Overview of chunking

1.1 Introduction

Chunking is a method for splitting sentences into smaller non-overlapping logical parts, or chunks. For example, from *Bird, Klein, and Loper (2009, pp 485)*:

```
[NP The morning flight] [PP from] [NP Denver] [VP has arrived]
```

The purpose of chunking can be to simplify parsing by chunking first, then joining the chunks to form a complete parse tree. Or it can be used to do some analysis that does not require a complete tree. Or it could be used as part of Named Entity Recognition.

There are several books that teach some techniques for chunking, amongst them Bird, Klein, and Loper (2009, (NLTK)) and Jurafsky and Martin (2009, (SLP)). NLTK teaches chunking based on the ConLL2000 task and Marcus and Ramshaw's notation, but uses regular expressions and n-grams as methods for extracting the chunks. NLTK then uses chunking for Named Entity Recognition or Relation Extraction. SLP also teaches chunking, or partial parsing as they sometimes call it, using either a variant of LR-parsing with hand-made rules, based on Abney, or a machine learning method using a training set.

This chapter will give an overview of some of the different methods of chunking and some of the theory behind chunking.

1.2 Stochastic NP chunk tagging

Church (1988) worked on automatic part-of-speech tagging of words, where he also did work on what we now can call NP chunking. Church's method was using an algorithm that used stochastic probabilities to assign POS-tags, where the training data is from a tagged Brown Corpus. Church achieved 95-99% accuracy using this method. He does not define chunking directly, nor use the word chunk, but he is using stochastic methods to find simple non-recursive noun pharses, where the training data has been made semi-automatically.

He created a program called PARTS, which have since been used to POS-tag sentences, and during that also insert chunk tags. His method of dividing

sentences into noun parses has been used by NLTK and Ramshaw and Marcus as an example of chunking.

1.2.1 Method

The method he used to generate the chunks is a simple stochastic method. Between each pair of POS-tags a probability of a chunk tag opening or closing at that point is calculated from the training data. When given a sequence of POS-tags the chunker simply enumerates all the possible combinations and then selects the most probable chunk structure as the chunk to return.

	AT	NN	NNS	VB	IN
AT	0	0	0	0	0
NN	.99	.01	.01	0	0
NNS	1.0	.02	.02	0	0
VB	1.0	1.0	1.0	0	0
IN	1.0	1.0	1.0	0	0

Table 1.1: Probability of Starting a Noun Phrase

	AT	NN	NNS	VB	IN
AT	0	0	0	0	0
NN	1.0	.01	.0	0	1.0
NNS	1.0	.02	.11	1.0	1.0
VB	0	0	0	0	0
IN	0	0	0	0	.02

Table 1.2: Probability of Ending a Noun Phrase

As can be seen in the tables 1.1 and 1.2, the chance for a chunk to start between a VB and a NN is 1.0, as the probability in (VB,NN) is 1.0. The probability of a chunk ending between a NNS and a NN is just 0.02, as (NNS,NN) is .02.

Examples of output from the PARTS program:

```
[A/AT former/AP top/NN aide/NN] to/IN [Attorney/NP/NP
General/NP/NP Edwin/NP/NP Meese/NP/NP] interceded/VBD
to/TO extend/VB [an/AT aircraft/NN company/NN 's/$
government/NN contract/NN] ,/, then/RB went/VBD
into/IN [business/NN] with/IN [a/AT lobbyist/NN]
[who/WPS] worked/VBD for/IN [the/AT defense/NN contractor/NN]
,/, according/IN to/IN [a/AT published/VBN report/NN] ./
[Tucker/NP/NP] said/VBD [the/AT investigation/NN]
involving/IN [Fairchild/NP/NP] had/HVD been/BEN going/VBG
on/IN [a/AT number/NN] of/IN [weeks/NNS] and/CC predates/VBZ
[last/AP week/NN 's/$ expansion/NN] of/IN [McKay/NP/NP 's/$
investigation/NP] to/TO include/VB [Meese/NP/NP] ./.
```

This is a sample of output from the PARTS program. The chunk tags are '[' and ']', while the notation after '/' are the POS-tag. The '/NP/NP'-tag are the marker for proper nouns. Things to note:

- The possessive marker "'" is split from the word it is connected to, for example *"company/NN 's/$"*.
- No chunk extends over a comma or other punctuation.
- The chunks are constructed as long as possible.
- A word can only be part of one chunk.

1.3 Partial parsing

Abney (1991) is one of the first to work on partial parsing of a whole sentence, which he called chunking, and not just extracting parts of a sentence. He based his work partly on measurements taken by Gee and Grosjan, who had measured several features during experiments, such as pause duration when reading. He argued for chunking as an effective way to parse sentences by using chunks before doing more complex analysis of the sentence.

Abney's chunks are defined from major heads, defined as content words except those that are between a function word and the content word that this function word selects. The root of the chunk's parse tree is the highest node for which the major head is the semantic head. Semantic heads can be related to syntactic heads:

- If the syntactic head is a content word, that word is also the semantic head.
- If the syntactic head is a function word, the content word that function word selects is the semantic head.

The parse tree of a chunk is a subgraph of the parse tree for the complete sentence, where the root of the tree is the semantic head.

Abney uses a non-deterministic LR-parser to determine the chunks, using a "simple toy grammar"[1], a small and simple grammar, followed by an LR-attacher to create the rest of the parse tree for the whole sentence. He does not, in his article, give the result from his chunker, nor his attacher, he just defines the method and algorithm.

Example, from the same article:

```
The effort to establish such a conclusion of
course must have two foci, the study of the rocks...
[DP [Det the] [NP [N effort]]]
[CP-Inf [IP-Inf [Inf-To to] [VP [V establish]]]]
[DP [Predet such] [Det a] [NP [N conclusion]]]
[CP [IP [AdvP [Adv of course]] [Modal will] [VP [V have]]]]
[DP [NP [Num two] [N foci]]]
[Comma ,]
[DP [Det the] [NP [N study]]]
[PP [P of] [DP [Det the] [NP [N rocks]]]]
...
```

[1] Abney's own name for his grammar

As Abney divides his parser into three parts, a scanner, a chunker, and an attacher, this example is the output from the chunker before it is passed on the attacher. As can be seen it consists of several trees, with phrase tags as nodes and words as leaves, here written in textual form, like lists in LISP. When we write out the chunk structure we see in the trees, removing the phrase tags and flattening it, and using the innermost phrase type as the chunk type, we get a structure similar to Church:

```
[NP The effort] [VP to establish] [NP such a conclusion]
[VP of course will have] [NP two foci] [,] [NP the study]
[NP of the rocks]
```

The name tag of the chunks, here NP or VP, is taken from the tag of the parent of the major head word, the last word in these trees. Several other naming conventions could be used, but this is used in most other works.

Abney is, compared to Church, more 'scientific' in his method of defining the chunks, and Abney also uses more types of chunks then Church, who only used NP-chunks. Their chunks cannot be compared directly, as their definitions of chunks are not similar. The output from Church's PARTS program makes generally shorter chunks, for example it always splits chunks at 'and' and '-', while Abney's chunks includes these when applicable.

1.4 Chunking using transformation-based learning

Ramshaw and Marcus (1995) base their work on chunks defined in the same style as Abney's chunks, and on extracting non-recursive NPs from the Penn Treebank, then trying to predict the chunks using a method called 'transformation-based learning' by Brill (1993).

Their training data was derived from the parses in the Wall Street Journal part of the Penn Treebank using two methods, creating two different training and test sets, one method finding the "BaseNPs", non-recursive noun parses, and one following Abney's model, called partitioned chunks. For Abney's chunks, they used N- and V- type chunks, where N corresponds to NP chunks with included prepositions, and V corresponds to the rest not included in the N-chunks. They handle possessives as a special case, where the possessive marker, ' 's ', was inserted into the next chunk, or starting the next chunk, as this might be useful for further processing.

Examples, BaseNP:

> *During [N the third quarter N], [N Compaq N] purchased [N a former Wang Laboratories manufacturing facility N] in [N Sterling N], [N Scotland N], which will be used for [N international service and repair operations N] .*
>
> *[N The government N] has [N other agencies and instruments N] for pursuing [N these other objectives N].*

The BaseNP-chunks were extracted from the parsed Treebank, by selecting the NPs that contained no other NPs, and also following the possessive rule above.

As they built the chunks from the Penn Treebank parses mostly directly, the rules for what constitutes a NP is the same as in the Treebank.

Examples, Abney's model:

> *[N Some bankers N] [V are reporting V] [N more inquiries than usual N] [N about CDs N] [N since Friday N]*
>
> *[N Indexing N] [N for the most part N] [V has involved simply buying V] [V and the holding V] [N stocks N] [N in the correct mix N] [V to mirror V] [N a stock market barometer N]*

Note that V-chunks are not strictly VP-chunks, and N-chunks are not strictly NP-chunks, but also include other parts. These were also built from the Treebank parses, but were subject to more transformations, such as the inclusion of the prepositions in PPs into NPs to form N-chunks, and the V-chunks being formed from several of the nodes in the parse tree. It also uses the possessive rule above.

1.4.1 Encoding

Marcus and Ramshaw did not use the method displayed in the examples above for encoding, but chose instead to use an encoding more suited to the algorithm they used. The encoding they used is {B,I,O} for BaseNP, where B marks the start of a chunk, I marks that the word is inside of the chunk, and O mark that the word is outside. For the partitioned chunks, they used {BN, N, BV, V, P}, where BN and BV marks the start of N and V type chunks, and N and V marks that the word is inside the chunks, and P marks that the word is punctuation and other special types. P is allowed to appear inside N and V chunks.

Encoding examples:

BaseNP:

> *During/O the/B third/I quarter/I , Compaq/B purchased/O a/B former/I Wang/I Laboratories/I manufacturing/I facility/I*

Abney's model:

> *Some/BN bankers/N are/BV reporting/V more/BN inquiries/N than/N usual/N about/BN CDs/N since/BN Friday/N*

A variant of the {B,I,O} encoding was used in the ConLL2000 chunking task. (Sang and Buchholz (2000))

1.4.2 Transformation-based learning

Transformation-based learning is a method for learning rules to assign correct features to objects in a set. The goal of this method is to make an ordered set of rules where each rule is applied in order to a set to transform it.

To make these transforming rules, the algorithm needs:

- An ordered collection of baseline rules. These rules are used to generate the working set from the training set, to make a starting position to generate rules from. These baseline rules are created manually beforehand.

- A working set, which at the start is created by stripping the training set of the features we are looking to generate, and then applying the baseline rules. This working set is gradually transformed during the algorithm to be more similar to the training set.

- A training set to score the generated rules when compared to the working set. This is the same set as used to generate the working set.

- A search space to constrain the generation of new transformation rules.

Having created these items, the method can begin to run.

During a run, all the possible rules, constrained by the search space, are generated for each object in the working set. These rules can be called the candidate rules. Each of these candidate rules are then applied, one rule at a time, to a copy of the current working set, and scored according to the effect each sole rule has on the working set when compared to the training set. The best scoring rule, which is the one that decreases the distance between the working and training set the most, is selected for inclusion in the output from the algorithm, and is also applied to the working set. The algorithm then runs again, until a stop value is reached, often a set number of runs. The output from the algorithm is an ordered set of rules, with the baseline rules at the top, followed by the first generated transformation rule. To set features on a new set, these rules have to be run in the order they were generated. An overview of the algorithm is in figure 1.1.

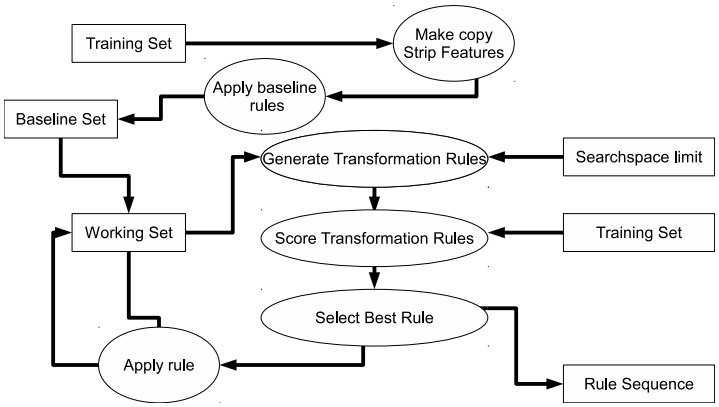


Figure 1.1: Transformation-based learning graph

1.4.3 Chunking using transformation-based learning

Marcus and Ramshaw use this method to generate the chunks. For the baseline working set, they choose to generate it by making rules that marked each word with the chunking tag the POS tag of that word had most commonly had in the training corpus. This working set was then used to make new transformation rules according to the algorithm. They also modified the algorithm somewhat by inserting some rules to make the generation go faster, and this might have influenced the result in a small way. The system was stopped when 500 rules

had been generated. The generated rules were then run against a test set of 50K words.

To measure the result, only the chunks that matched exactly the chunks in the test set were counted, that is, only whole and correct chunks were counted, not part of chunks. The results are in table 1.3.

Size of training	BaseNP		Abney's	
	Recall	Precision	Recall	Precision
Baseline	81.9	78.2	60.0	47.8
50K	90.4	89.8	86.6	85.8
100K	91.8	91.3	88.2	87.4
200K	92.3	91.8	88.5	87.7
950K	93.5	93.1	Not run	Not run

Table 1.3: Results from Marcus and Ramshaw

Of note here is that the BaseNP chunks seems to be easier to predict than partitioned chunks, as expected, as the BaseNP chunks are generally simpler then the partitioned chunks. Also, the results get significantly better with a larger training set. Comparing the BaseNP results to Church's results is possible, with Church having 95% to 99% correct, however neither of the two chunks styles are alike, BaseNP chunks and the chunks used by Church are not the same, and the definition of "correct" is also dissimilar.

1.5 Conll2000 Chunking Task

1.5.1 Introduction

The Conll2000 chunking task was a test of several different methods for making chunks by several researchers. Sang and Buchholz (2000), who made the materials used by the other researchers, used a method similar to Ramshaw and Marcus (1995), extracting chunks from the parse tree in the Penn Treebank to make training data, but Sang and Buchholz used other types of chunks. The generated training data were then given to the other researchers in the task.

In table 1.5 is the chunks extracted from the Penn Treebank. Note that this method of extracting, as it was with Marcus and Ramshaw's extractions, are sensitive to errors in the parse tree in the Treebank, and will therefore have some errors introduced that way.

The training and test set was part of the Penn Treebank, specifically section 15-18 of the WSJ for training data, and section 20 for the test data, about 210K words for the training set, and 50K words for the test set.[2] For encoding, a similar system as Marcus and Ramshaw were used, B-X, I-X, and O, where X is the type of chunk. Also, to make the test more realistic, a POS tagger by Brill (1994) were used to tag each word in the training set, instead of using the Penn Treebank tags. This is an example of the training and test set:

```
He          PRP  B-NP
reckons     VBZ  B-VP
```

[2]The same sets as Ramshaw and Marcus used.

Name of chunk (Occurrences)	Description
NP (55081 - 51%)	Uses very similar chunks as Ramshaw and Marcus's BaseNP chunks, also with the possessive split from the chunk. If an ADJP is part of the NP, it becomes part of the NP chunk.
VP (21467 - 20%)	VP chunks are constructed by joining all VPs in the tree that are joined, that is, on the form *(VP . (VP . (VP .)))* ADVP becomes part of the VP chunk if it is in front of the main verb
ADVP (4227 - 4%)	ADVP chunks are the ADVPs in the Treebank, except when they are inside an ADJP or VP and in front of the main verb. In that case they are included in the VP or ADJP chunk respectively. If the ADVP contains a NP, it becomes two chunks (NP and ADVP)
ADJP (2060 - 2%)	ADJP chunks are the ADJP in the Treebank, except if they are part of a NP. ADJP that contains an NP is split into two chunks
PP (21281 - 20%)	PP chunks usually just consist of the single or multiword preposition, with the POS-tag IN
SBAR (2207 - 2%)	SBAR chunks are like PP chunks and are usually a single or multiword complemetizer.
CONJP (56 - 0%)	CONJP chunks can be more then one word. One-word conjunctions in the Treebank are not annotated as CONJP, and are therefore not marked as CONJP chunks
PTR, INTJ, LST, UCP (599 - 1%)	These chunks are the same as their Treebank parts
Things not marked as chunks	Some parts are not part of any chunks, mostly punctuation, and the word 'not' in special cases

Table 1.5: Types of chunks extracted from the Penn Treebank

```
the         DT   B-NP
current     JJ   I-NP
account     NN   I-NP
deficit     NN   I-NP
will        MD   B-VP
narrow      VB   I-VP
to          TO   B-PP
only        RB   B-NP
#           #    I-NP
1.8         CD   I-NP
billion     CD   I-NP
in          IN   B-PP
September   NNP  B-NP
.           .    O
```

1.5.2 Results:

Researcher	Method	$F_{\beta=1}$	Precision	Recall
Kudoh and Matsumoto	231 support vector machine classifiers	93.48	93.45	93.51
Van Halteren	4 WPDV taggers and a memory-based tagger	93.32	93.13	93.51
Zhou, Tey and Su	Hidden Markov Models	92.12	91.99	92.25
Dejean	ALLiS system	92.09	91.87	92.31
Koeling	Maximum-entropy learner	91.97	92.08	91.86
Osborne	Modified Ratnaparkhi's maximum- entropy POS tagger	91.94	91.65	92.23
Veenstran and Bosch	Memory-based learning algorithm	91.54	91.05	92.03
Pla, Molina and Prieto	Finite-state Markov Models	90.14	90.63	89.65
Johansson	Context-sensitive and context-free transformation rules	87.23	86.24	88.25
Vilain and Day	Alembic parser using transformation based rules	85.76	88.82	82.91
Baseline	Most common chunk tag from POS-tag	77.07	72.58	82.14

Table 1.6: Results from the ConLL2000 task

For the ConLL2000 task, the training and test sets detailed above were given to ten groups of researchers. They then used different methods and algorithms to generate the chunks. For evaluating the chunkers, an $F_{\beta=1}$ score were used, with a chunk only counting if the whole chunk matched, same as with Marcus and Ramshaw.

The results, displayed in table 1.6, with a baseline being generated by assigning each word the most frequent chunk tag according to the POS-tag of that word, ranged in the order of an $F_{\beta=1}$ value from 85.76 to 93.48, with the baseline being 77.07. The top results all used a combination of several methods, with a voting step at the end combining the results to select the chunk tag. The two best scoring methods were one method using 231 support vector machine classifiers by Kudoh and Matsumoto (2000) and one using four weighted probability distribution classifiers by Halteren (2000), scoring significantly better than the rest.

Again, these chunks are too dissimilar to both Marcus and Ramshaw's chunks and to Church's chunks to compare directly, except the NP-chunks that are nearly similar except some minor differences. We can however note that the precision and recall of the two best performing methods are very close to the best result from Ramshaw and Marcus on the BaseNPs. Also, the three top performing methods are all combination methods, running several methods in parallel, and selecting the top voted result as the final result.

1.6 Conclusion

As can be seen, several different methods have been used both to generate chunks, and make training data for the methods. Church worked on generating NP-like chunks using stochastic data trained on semi-manually created training data. He did not however specify what criteria he used to create the training data. Abney creates a method for determine some types of chunks, based partly on the way we divide sentences when we talk, and partly on an attempt to simplify parsing. He does not give any results from his work, only the method he used. Both Ramshaw and Marcus and the ConLL2000 task uses somewhat similar chunks, most with NP-chunks, and both extract those chunks from the Penn Treebank. They are however both bound by the parsing in the Penn Treebank, and will replicate any errors that exist there, in addition to being constrained by the choices done in that parsing.

There are also two different methods for encoding the chunks. Church uses an encoding that is connected to the sentence, with the chunk data being between the words in the sentence. This can be compared to the encoding used by the others, where the chunk information is connected to the words individually, and each word having its own chunk tag. The choice of encoding is important regarding the choice one has in methods for generating chunks. When using chunk tags connected to each word, one does not have to close the chunk explicitly, and if an 'inside chunk' tag is encountered without a preceding 'begin chunk' tag of the correct type, the 'inside chunk' tag can be assumed to start a new chunk. This compares to the sentence connected chunk data, where each chunk must be closed and opened correctly.

When it comes to generating chunks on the test data, the results and methods used cannot be compared directly, as all use different types of chunks. Church uses a quite simple stochastic method to generate simple NPs, and gets quite good results, in the 95% - 99% range. Abney, as noted above, does not give any result from his method, a LR parser and a LR attacher. Marcus and Ramshaw uses transformation-based learning to generate chunks, and gets similar results as the ConLL2000 task, but their chunks are not similar. However,

the ConLL2000 task creates the same chunks using many different methods. From that it can be seen that the best methods they used were many methods running in parallel, with the best result from a method using 231 support vector machines (Kudoh and Matsumoto, 2000), with a $F_{\beta=1}$ score of 93.48.

Chapter 2

Converting a dependency graph to a chunk structure

2.1 Introduction

A dependency graph is a graph where the words are connected by dependencies to each other. With analysis of the dependency graph and POS information in the sentence, it is possible to use a dependency graph to create a chunk structure for the same sentence. This chapter describes the process used to create a chunk structure using the Norwegian Dependency Treebank.

2.2 What is a dependency graph?

A dependency graph is a structure where each word is connected by a dependency between two single words. A word is either a head or a dependent, where the dependent depends on, or is dominated by, the head. This can for example be:

- Subject relation: *Mary* hit John
- Object relation: John hit *Mary*
- Subject relation: *The terrifyingly huge and monstrously gigantic Mary* hit John
- Object relation: John hit *the gray and orange Mary*

In the first two cases both 'John' and 'Mary' depends on the verb 'hit'. In the third case 'Mary' is a subject dependent on 'hit', while all the words in 'The terrifyingly huge and monstrously gigantic ' are dependent on either 'Mary' or a word that is, through the other words, dependent on 'Mary'. The same is in the fourth case, where 'Mary' is an object dependent on 'hit', while all the words in 'the gray and orange' are dependent on either 'Mary' or a word that is, through the other words, dependent on 'Mary'. In the above example, the chunk tags would be:

- NP[Mary] VP[hit] NP[John]

- NP[John] VP[hit] NP[Mary]
- NP[The terrifyingly huge and monstrously gigantic Mary] VP[hit] NP[John]
- NP[John] VP[hit] NP[the gray and orange Mary]

A more complex dependency structure can be seen in figure 2.1.

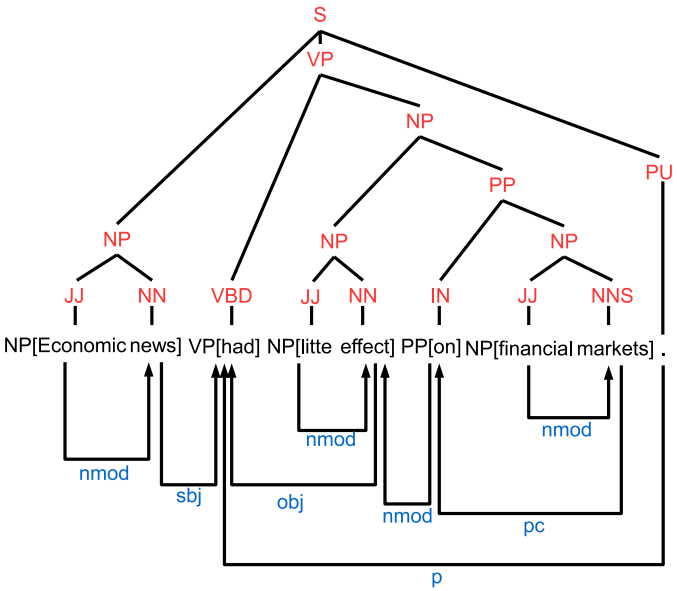


Figure 2.1: This is a combined dependency and phrase structure graph with chunks marked.

In figure 2.1 we can see that each word has a single dependency head, and that there is a chain linking all the words up to the root of the tree, 'had'. We also see that the chunk structure here corresponds to the lower levels of the parsed structure, as is intended.

In addition, each dependency link also signals what the relationship between the words is. For example, using figure 2.1, the link $had \overleftarrow{< ojb >} effect$ and $effect \overleftarrow{< nmod >} little$. Here, "effect" is the object of the verb "had", while "little" modifies "effect". There are many such dependency relations, but the relations used depends on the specific dependency structure used.

2.3 From dependency structure to a chunk structure

As can be seen in figure 2.1, the NP chunks correspond to the dependency links of type 'nmod'. This means that in this spesific sentence we could have extracted the NP chunks by simply finding each NN and using the dependency link to determine that the JJ and NN should be grouped together in a NP chunk. The resulting chunk structure is shown on the figure. The single-word chunks are simply determined from that word's part of speech.

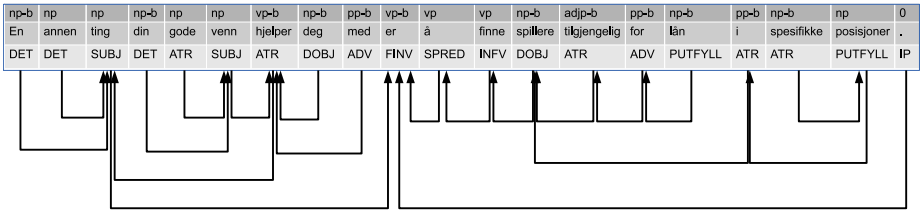


Figure 2.2: This is a sentence from the corpus, showing the dependency links and the chunk tags generated by our rules

Figure 2.2 shows a more complex example. Here we can see that each chunk consists of a dependency head and some of its direct dependents and transitive dependents. This is further described in 2.5.1. The head of the phrase does not need to be the same as the head of the dependency chain, but it will often be the case that it is. We can also see that we do not need to look at the type of the dependency link (e.g. 'DET, 'SUBJ,...), only the link itself. To make the chunk in this case is the same as for figure 2.1. First, find the head of the chunk, and then follow its dependency links, both upwards and downwards, until a chunk have been made. The rules we are using are described in 2.5.1. Note that this method of finding chunks corresponds to the definition used by Abney, see Abney (1991), that chunks are determined by finding the semantic head of a word.

2.4 The Norwegian dependency treebank

We are generating chunks using the Norwegian Dependency Treebank (*Norsk dependenstrebank* (2014)). This is a dependency Treebank for the Norwegian languages Bokmål and Nynorsk; we are using the Bokmål part. The Treebank is annotated with part of speech, morphological features, and, of course, dependency links. It consists of sentences taken from newspaper articles, parliamentary meetings, governmental reports, and private blogs, with most of the sentences in the Treebank coming from newspaper articles. The dependency graph in the Treebank follows these rules:

- No unconnected words. This means that all the words in the sentence are connected to that sentence's dependency graph, with a single head on the top. For our usage, this means that we can at all words use the dependency graph. Without this feature, we would need to account for words that are not connected, both when chunking that word, and when determining the chunks of adjacent words. This would have made our analysis harder.

- No empty nodes. In the dependency graph, every node of the dependency graph is connected to a word. This will not affect our conversion much, but it makes it a bit easier to analyze the dependency structure.

- Only unique dependencies. Each word is dependent on one, and only one, word. Except the head of the sentence, which does not have a dependency

head. This just means that we have to depend solely on the main dependency structure. See Haug (2010) for examples of dependency structures with secondary dependencies.

- Crossing dependencies. In the dependency structure, a dependent may be separated from the other dependents on the same head by other words not in the dependent group. Chunking works differently, no part of a chunk is allowed to cross over other words, and a chunk must be linearly constructed. We are using these properties heavily to construct our chunks, as we use a word not connected directly to none of the words in the chunk we are constructing to signal an end of the chunk we are constructing.

2.5 Converting the NDT to a chunk structure

We are using the Norwegian Dependency Treebank to create a chunk structure for the same Treebank. To do so, we have made a set of rules for analyzing the information in the Treebank and creating the chunk structure. The rules are described below. Note that these rules have been made using version 0.3 of the NDT. Some changes can be expected for the higher versions of the NDT.

2.5.1 The Rules

2.5.1.1 Base of the algorithm

The statements below are what we are building the rules on. These rules are meant to be applied to one sentence at a time.

- A chunk must be continuous without any words that are not part of the chunk inside it. This means that when searching for words to include in the chunk, if we come to a word that does not fit, we can stop the search there. This comes for example when the determiner is separate from the noun and there is a word not part of the chunk between them. An exception to this rule may be to allow a chunk to stretch over punctuation and other such marks, but we have not allowed this, except in proper names, e.g. NP[Peter 'The Great' of Russia]

- The head of a chunk is most often the rightmost word in the chunk. This means that to find a chunk we search for the headword and then build the chunk by going backwards in the sentence from that headword. This does not always work, for example with full names, where the head might be earlier in the chunk. However, we can still build the chunk from finding the rightmost word that is a possible candidate for a headword and going backwards.

- We can observe that words that are adjacent and directly connected in the dependency structure are also in the same chunk. This means that when we come to a word, when going backwards in the sentence, we determine if it should be in the same chunk by checking if this word is connected to the headword of the chunk through the dependency tree.

- Conversely, when we have two words that might be in the same chunk when looking at the POS tags, but they are not connected in the dependency

structure, then these two words are not in the same chunk. This means that when we come to such a pair of words, we do not include them in the same chunk

- We also have some absolutes:
  - A noun or pronoun can never be part of any chunk except a NP chunk
  - A verb can never be part of any chunk except a VP chunk
  - A preposition can never be part of any chunk except a PP chunk
- And some priorities[1]:
  - An ADJP chunk will be part of the NP chunk if it is in front of the NP chunk
  - An ADVP chunk will be part of the VP chunk if it is in front of the VP chunk
  - An ADVP chunk will be part of the ADJP chunk if it is in front of the ADJP chunk

The algorithm for generating chunks from the dependency gold standard is based on the above statements.

2.5.1.2 Symbols used and general flow

- In this description we are using the following symbols:
  - W is the current position in the sentence in the forward search
  - WB is the current position in the sentence in the backward search
  - C is the chunk currently being filled. This chunk is a set consisting of positions.
  - C_W is the chunk of the word at position W. This is a set consisting of positions.
  - C_{W+X} is the chunk of the word X positions from W. This is a set consisting of positions.
  - $Word_{W+X}$ is the word X positions from W
  - H_W is the dependency head of the word at position W
  - H_{W+X} is the dependency head of the word X positions from W
  - D_W is the set of dependents of the word at position W
  - D_{W+X} is the set of dependents of the word X positions from W
  - $Link_W$ is the type of the word at position W's dependency link
  - $Link_{W+X}$ is the type of the word X positions from W's dependency link
  - S is the current sentence
  - T_W is the part of speech of the word at position W. See table 2.1

[1] These are from Sang and Buchholz (2000)

- T_{W+X} is the part of speech of the word X positions from W
- $Morphology_W$ is the morphology of the word at position W. It is a set.

- The algorithm goes through the sentence several times, one time for each chunk type, in the priority order listed below. If a word is already assigned a chunk, a lower priority chunk will not overwrite it, except step 7.

  1. NP
  2. VP
  3. PP
  4. ADJP
  5. ADVP
  6. Other chunk classes, KONJP, SBUP, INTERJP. These classes do not overlap at all, and therefore have no need for prioritizing
  7. We also do a cleanup round at the end, to catch words not assigned to a chunk, or leftovers. This is mostly either determinants or incomplete words. These are all assigned to a NP chunk. This step may overwrite existing, higher priority, chunks.

Part of speech	Mark in corpus
Noun	subst
Verb	verb
Preposition	prep
Determiner	det
Adjective	adj
Adverb	adv
Infinite mark	inf-merke
Conjuntion	konj
Subordinating conjunction	sbu
Interjection	interj
Pronoun	pron
Conjunction	konj
Incomplete	ufl

Table 2.1: Parts of speech. In the corpus, there are also some that are clearly wrong, and only appears once

2.5.1.3 The Algorithm

```
NP Search:
for each word in sentence:
  if rule1 is true:
    apply rule1
    search backwards when stated
  else if rule2 is true
```

```
    apply rule2
    search backwards when stated
  ...
VP Search:
for each word in sentence:
  if rule1 is true:
    apply rule1
    search backwards when stated
  else if rule2 is true
    apply rule2
    search backwards when stated
    ...
```

This is the outline of the algorithm. Once for each type of chunk, in the order listed below, the above is run once for each word in the sentence. It is also run once for 'misc', and once for the leftovers. If a word already assigned to a chunk is reassigned, that word is removed from the old chunk. A chunk that is empty, containing zero words, is removed.

1. **NP Search**

   At start, set W to the position of the first word in S. When a rule matches, start from the first rule again, and set W to $W + 1$

   (a) If $T_W = subst$, and $D_W \cap C_{W-1} \neq \emptyset$ or $H_W \in C_{W-1}$, and C_{W-1} is an NP chunk, add W to C_{W-1}

   - This is mostly the case with two nouns in a row. Always used in proper names.

   (b) If $T_W = det$, and $H_W \in C_{W-1}$, add W to C_{W-1}

   - This catches the cases where the determiner is after the noun.

   (c) If $T_W = subst$ or $T_W = pron$, start a new C of type NP and add W to it. Set WB to $W-1$, and start looking backwards in the sentence.

   i. If $C_{WB} \neq \emptyset$, and it is not of type NP, that is WB has a chunk that is not an NP chunk, stop searching backwards.

   ii. If $T_{WB} = subst$, and $H_{WB} \in C_W \neq \emptyset$ or $D_{WB} \cap C_W \neq \emptyset$, add WB to C, then set WB to $WB - 1$

   iii. If $T_{WB} = konj$, and $H_{WB} \in C_W$, and $D_{WB-1} \cap C_W \neq \emptyset$ or $H_{WB-1} \in C_W$, add WB to C, then set WB to $WB - 1$

   A. This rule join phrases like "bil og buss", where all the words in the phrase is dependent in some way to each other, into a chunk "NP[bil og buss]". If they are not dependent on each other, the conjunction "og" should be a KONJP chunk instead. This is done by rule 6b.

   iv. If $T_{WB} = adv$, and $H_{WB} \in C_W$, add WB to C, then set WB to $WB - 1$

   v. If $T_{WB} = adj$, and $H_{WB} \in C_W$ or $D_{WB} \cap C_W \neq \emptyset$, add WB to C, then set WB to $WB - 1$

   vi. If $T_{WB} = det$, and $H_{WB} \in C_W$ or $D_{WB} \cap C_W \neq \emptyset$, add WB to C, then set WB to $WB - 1$

vii. If none of the above is true, stop searching backwards

2. **VP Search**

   (a) If $T_W = verb$, and C_W is not of type NP, start a new C of type VP and add W to it. Set WB to $W-1$, and start looking backwards in the sentence.

      i. If C_{WB} exists, and it is not of type VP, stop searching backwards
      ii. If $T_{WB} = verb$, and $H_{WB} \in C_W$ or $D_{WB} \cap C_W \neq \emptyset$, add WB to C, then set WB to $WB-1$
      iii. If $T_{WB} = adv$, and $H_{WB} \in C_W$, and $Link_{WB} = ADV$, add WB to C, then set WB to $WB-1$
      iv. If $T_{WB} = inf-merke$, and $H_{WB} \in C_W$ or $D_{WB} \cap C_W \neq \emptyset$, add WB to C, then set WB to $WB-1$
      v. If none of the above is true, stop searching backwards

3. **PP Search**

   (a) If $T_W = prep$, and C_W is not of type NP or VP, start a new C of type PP and add W to it. Set WB to $W-1$, and start looking backwards in the sentence.

      i. If C_{WB} exists, and it is not of type PP, stop searching backwards
      ii. If $T_{WB} = prep$, and $H_{WB} \in C_W \neq \emptyset$ or $D_{WB} \cap C_W \neq \emptyset$, add WB to C, then set WB to $W-1$
      iii. If none of the above is true, stop searching backwards

   (b) If $T_W = det$, and W is not in a chunk, and $H_W \in C_{W-1}$, and C_{W-1} is of type PP, add W to C_{W-1}

4. **ADJP Search**

   (a) If $T_W = adj$, and C_W is not of type NP or VP or PP, start a new C of type ADJP and add W to it. Set WB to $W-1$, and start looking backwards in the sentence.

      i. If C_{WB} exists, and it is not of type ADJP, stop searching backwards
      ii. If $T_{WB} = adj$ or $T_{WB} = adv$ or $T_{WB} = det$, and $H_{WB} \in C_W$ or $D_{WB} \cap C_W \neq \emptyset$, add WB to C, then set WB to $WB-1$
      iii. If none of the above is true, stop searching backwards

5. **ADVP Search**

   (a) If $T_W = adv$, and C_W is not of type NP or VP or PP or ADJP, start a new C of type ADVP and add W to it. Set WB to $W-1$, and start looking backwards in the sentence.

      i. If C_{WB} exists, and it is not of type ADVP, stop searching backwards
      ii. If $T_{WB} = adv$, and $H_{WB} \in C_W$ or $D_{WB} \cap C_W \neq \emptyset$, add WB to C, then set WB to $W-1$

iii. If none of the above is true, stop searching backwards

6. **Misc search**

   All these words are single-word chunks

   (a) If $T_W = sbu$, and $Link_W = SBU$ or $Link_W = SBUREL$ or $Link_W = SBUREL$ or $Link_W = SUBJ$ or $Link_W = PUTFYLL$ or $Link_W = ADV$, start a new C of type SBUP and add W to it

   (b) If $T_W = konj$, and C_W is not in a chunk, start a new C of type KONJP and add W to it

   (c) If $T_W = interj$, and C_W is not in a chunk, start a new C of type INTERJP and add W to it

7. **Leftovers search**

   After doing the above cases, the algorithm sets chunks on the 'leftovers', using slightly more analysis. This is done in the same way as the normal rules.

   (a) First, we have the determinants. Either these are added to the closest correct chunk, or a new chunk is made for them.

      i. If $T_w = det$ and W is not in a chunk, do one of the following

         A. If $H_W = H_{W+1}$, add W to C_{W+1}

            - This case is for catching determiners in front of an adjective, where both has the same head, and should therefore be in the same chunk.

         B. If C_{H_W} is a NP chunk, create a new C of type NP, and add W to it.

            - This is to catch cases where the determiner is dependent and determining a word that is inside quotation marks or parentheses. For example:

              `NP[sin] “ NP[onkel Mulle] ”`

         C. If $W = H_{W-1}$, and C_{W-1} is a NP chunk[2], add W to C_{W-1}

         D. If $T_{W+1} = det$, and $T_W = det$, and $Link_{W+1} = DET$ and $Link_W = DET$, and W_{W+1} is not in a chunk, create a new C of type NP, and add W and W_{W+1} to it

            - This case is to catch words that behave like substitute subjects and should therefore be a NP chunk.

         E. If $H_W = W + 1$ and $T_{W+1} = det$, and $W + 1$ is not in a chunk, create a new C of type NP, and add W and $W + 1$to it

            - This case is to catch words that behave like substitute subjects and should therefore be a NP chunk. Example:

              `PP[med/PREP] NP[hans/DET eget/DET] ./$`

         F. If $H_{W+1} = W$, and C_{W+1} is a PP chunk, and $Link_{W+1} = ATR$, add W to C_{W+1}

[2] Note that $W - 1$ should always be a noun, but don't need to be.

- This is to catch determinant that determine prepositions.

G. If $Link_W = SPRED$ or $Link_W = OPRED$ or $Link_W = DOBJ$ or $Link_W = ADV$ or $Link_W = KOORD - ELL$ or $Link_W = SUBJ$ or $Link_W = KOORT$ or $Link_W = IK$ or $Link_W = PUTFYLL$ or $Link_W = DET$, create a new C of type NP, and add W to it.

- This is to catch cases where there is a single determiner disconnected from its owner. These are made into NP chunks, as they behave as substitute subjects or objects.

(b) Then we have the words that have been marked as incomplete. This mark is also used sometimes for abbreviations.

i. If $T_w = ufl$ and W is not in a chunk, do one of the following

A. If C_{H_W} is a PP chunk, and $H_W = W - 1$, add W to C_{W-1}.

- This is to catch words that belong in the chunk right before them, but have been analyzed to be of type 'ufl'

B. If C_{W+1} is a NP chunk, and $D_W \cap C_{W+1} \neq \emptyset$, add W to C_{W+1}

- In addition, If C_{W-1} is a ADJP chunk, and $D_W \cap C_{W-1} \neq \emptyset$, then add C_{W-1} to C_{W+1}.
- Also, if $T_{W-1} = det$, and $H_{W-1} = W$, add $W-1$ to C_{W+1}.
  - This is one of the few rules that change another chunk.
  - These rules are to join NPs that have been split by an 'ufl' word. This happens when you have a structure like 'den internasjonale avis- og nettflora' where 'avis-' has been marked as an 'ufl'

C. If $T_{H_W} = ufl$, and $T_{W-1} = konj$, and $H_{W-1} = W$, create a new C of type NP and add H_W and W and W_{W-1} to it.

- This case might also change an already existing chunk.
- This rule is to join words that have been analyzed to be expressions, marked with 'ufl' and a conjunction, and therefore should be an NP chunk together. Example,

```
Brask og bram (1415)
```

D. If $T_{H_W} = subst$, add W to C_{H_W}.

E. If $H_{W-1} = W$, and C_{W-1} is of type ADJP, add W to C_{W-1}.

- This is to catch words marked 'ufl' where the word belongs in the previous word's chunk

(c) Next, infinite marks

i. If $T_W = inf - merke$ and W is not in a chunk, and one of D_W is in VP chunk, create a new C of type VP and add W to it.

- This happens when the infinite mark and its word is separated by quote marks or its like.

(d) And searching for proper names is last.

i. If $T_W = subst$, and $prop \in Morphology_W$, set X to 1

A. If $H_{W+X} = W$, and $Link_{W+X} = FLAT$, create a new C of type NP, and add W and W_{W+X} to C. Set Y to 1, increase X by 1, and run this rule again.

B. If $Y = 1$ and $T_{W-1} = konj$ and C_{W-1} is of type NP, add all members of C_{W-1} to C_W.
   - These are to add proper names, like 'Den Norske Kirke', to a single NP chunk

The above rules leave only punctuation marks and words the dependency analysis has marked as of unknown type, $T_W = ukjent$. There are also misspelled types, like $T_W = dem$[3] or $T_W = sybst$[4]. These are ignored, and left without chunks.

2.5.1.4 Genitive split

When the above is done, we go through all the chunks, and split the chunks into two parts if a word has the genitive marker in the morphology, 'gen'.

```
NP[Thorvald Hansens store hus]
```

Gets split into

```
NP[Thorvald Hansens] GNP[store hus]
```

This is done to retain the ownership information stored in the dependency analysis. The new chunk tag 'GNP' is used, standing for 'Genitive NP'. This is also done in the English chunks by Sang and Buchholz (2000) and Ramshaw and Marcus (1995), although in English the word in the genitive case is itself split

```
NP[Thorvald Hansen's big house]
```

Into

```
NP[Thorvald Hansen] NP['s big house]
```

This method of splitting the chunks containing a genitive case is less possible in Norwegian, as Norwegian do not use the ' ' marker. We therefore use the tag GNP instead to mark the possessed noun. We also do not create a new chunk in the cases where nothing is owned[5]

This splitting of genitive chunks is done only by using the genitive marker generated by the dependency analysis. We do not do any extra analysis.

2.5.2 Considerations

As there do not exist a readymade chunk structure for Norwegian, nor a set of rules to follow, we had to create these rules using the English chunks by Church and those used by the ConLL2000 task as guidelines. We also had to consider what was possible to extract from the dependency structure. Some of these considerations are presented here.

[3] 'Det' I think

[4] 'Subst' perhaps?

[5] In the english examples, constructions like NP[Hansen] NP['s] are possible

2.5.2.1 Chunks are split by punctuation- and other marks.

We choose to nearly always split the chunks at punctuation, quote, comma, and other marks. The only exception is when the marks have been defined having the dependency link type of FLAT. Then they are included in the NP chunk. See for example:

```
NP[Espen « Shampo » Knutsens]
```

Where all the word's dependency heads are to "Espen", and their dependency link is of type FLAT. They are then considered a single NP chunk. This is done by rule 7(d)i.

2.5.2.2 Chunks over or split by the word "og"

The word "og", "and", is a conjunction. Either this word should be part of a single NP-chunk, or it should split and be a CONJP-chunk by itself. We choose to follow the dependency structure closely. In cases like

```
NP[Finanskrisen og dens ettervirkninger]
```

The word "og"'s dependency head is "ettervirkinger", and "Finanskrisen" is "ettervirkningers"'s head. "dens"'s head is "ettervirkninger". Since they are connected, they are put in the same chunk. In other cases, like

```
NP[pågangsmot] CONJP[og] NP[arbeidsevne]
```

The word "og"'s dependency head is "arbeidsevne", but "arbeidsevne"'s head is not "pågangsmot", nor is "pågangsmot"'s head "arbeidsevne". This means that they are not in the same chunk

2.5.2.3 Genitive split

We split the owned word or object from its owner into its own chunk. This is to save the ownership information available in the dependency Treebank, and to correspond to the same method used in the ConLL2000 task. More info is in 2.5.1.4

2.6 An example of applying the rules

This will be a complex example of a running of the rules over a sentence from the corpus, using the sentence

```
Utvalget har fulgt opp dette, og også beskrevet de viktigste
konsekvensene som de ulike alternativene innebærer for staten,
Den norske kirke og andre tros- og livssynssamfunn
```

Chunked:

```
NP[Utvalget] VP[har fulgt] PP[opp] NP[dette] , CONJP[og]
VP[også beskrevet] NP[de viktigste konsekvensene] SBUP[som]
NP[de ulike alternativene] VP[innebærer] PP[for] NP[staten],
NP[Den norske kirke] CONJP[og] NP[andre tros- og livssynssamfunn].
```

The rules are using the information in table 2.2:

Table 2.2: Sentence 15805

Id	Word[6]	Type	Head	Dep. link	Genitive	Morphology
1	Utvalget	subst	2	SUBJ	False	NA
2	har	verb	0	FINV	False	NA
3	fulgt	verb	2	INFV	False	NA
4	opp	prep	3	ADV	False	NA
5	dette	pron	3	DOBJ	False	NA
6	,	<komma>	3	IK	False	NA
7	og	konj	9	KONJ	False	NA
8	også	adv	9	ADV	False	NA
9	beskrevet	verb	3	KOORD	False	NA
10	de	det	12	DET	False	NA
11	viktigste	adj	12	ATR	False	NA
12	kons.	subst	9	DOBJ	False	NA
13	som	sbu	17	SBUREL	False	NA
14	de	det	16	DET	False	NA
15	ulike	adj	16	ATR	False	NA
16	altern.	subst	17	SUBJ	False	NA
17	innebærer	verb	12	ATR	False	NA
18	for	prep	17	ADV	False	NA
19	staten	subst	18	PUTFYLL	False	NA
20	,	<komma>	19	IK	False	NA
21	Den	subst	19	KOORD	False	prop
22	norske	adj	21	FLAT	False	NA
23	kirke	subst	21	FLAT	False	NA
24	og	konj	26	KONJ	False	NA
25	andre	det	26	DET	False	NA
26	tros-	ufl	19	KOORD	False	NA
27	og	konj	28	KONJ	False	NA
28	livssynss.	subst	26	KOORD	False	NA
29	.	<punkt>	2	IP	False	NA

2.6.1 NP-search

The algorithm begins by examining the first word in the sentence, and tries to match one of the NP rules to the word. Rule NP-1c matches the first word, starting a NP chunk, and beginning a backward search. This gives:

```
NP[Utvalget] har fulgt...
```

Since this is the first word, the search backwards stop immediately.

The algorithm then continues testing word for word, testing each word if one of the NP-rules matches. At word 5, rule NP-1c matches, starting a chunk, and beginning a backward search. This gives:

```
...opp NP[dette] , og...
```

Then the algorithm searches backwards, first at word 4. Since none of the backward rules under rule NP-1c matches, it stops.

At word 10, rule NP-1b partially matches. However, since the head of word 10, word 12, is not in the chunk of word 9, rule NP-1b do not match.

The next matching word is word 12, matching rule NP-1c. This gives:

```
...beskrevet de viktigste NP[konsekvensene] ...
```

The backwards search starts at word 11. It is an adjective, 'adj', and word 11's head is word 12. This matches rule NP-1(c)v, and the search then tests word 10. Word 10 is a determiner, 'det', and word 10's head is word 12, so it matches rule NP-1(c)vi. Word 9 is a verb, matching none of the rules, and stopping the search. This gives:

```
...beskrevet NP[de viktigste konsekvensene] som...
```

Word 14 only partially matches rule NP-1b. However, word 16 matches rule NP-1c, starting a chunk. The backwards search finds that word 15 matches rule NP-1(c)v and word 14 matches rule NP-1(c)vi. Word 13 does not match any rule, stopping the search. This gives:

```
...som NP[de ulike alternativene] innebærer...
```

Word 19 matches rule NP-1c, and word 18 does not match any backward search rule. This gives:

```
...for NP[staten], Den...
```

Word 21 matches NP-1c, starting a NP chunk. However, word 20 is a comma, that does not match anything, stopping the backwards search. This gives:

```
...NP[staten], NP[Den] norske...
```

Word 23 matches NP-1c, starting a NP chunk. However, since word 22, while an adjective and partially matching rule NP-1(c)v, has its head at 21, word 22 is not added to the chunk, stopping the backwards search. Note that these three words, 'Den norske kirke', should have been a single NP. This is fixed later in the example. This gives:

```
...norske NP[kirke] og...
```

Word 28 matches NP-1c, starting a NP chunk. Word 27 is a conjunction, matching rule NP-1(c)iii. However, word 26 is an unfinished word, matching nothing and stopping the backwards search. This gives:

```
...  tros- NP[og livssynssamfunn].
```

2.6.2 VP-search

The algorithm starts again from the first word, with the sentences chunked with the NP-chunks as input. Word 2 matches rule VP-2a, starting a VP chunk, and beginning a backward search. This gives:

```
NP[Utvalget] VP[har] fulgt...
```

Since word 1 has a chunk that is not a VP chunk, the backwards search stops.

Word 3 matches rule VP-2a, starting a VP chunk, and beginning a backward search. This gives:

```
NP[Utvalget] VP[har] VP[fulgt]...
```

Since word 2 is a verb and word 3 is dependent on word 2, rule VP-2(a)ii matches word 2. This removes the current VP-chunk from word 2, and adds word 2 to the VP-chunk of word 3. Word 1 is a NP-chunk, and it stops the backward search. This gives:

```
NP[Utvalget] VP[har fulgt]...
```

The next matching word is word 9. Word 9 is a verb that matches rule VP-2a, and the backwards search finds word 8. Word 8's head is word 9, and word 8's dependency link is of type ADV. Word 8 matches rule VP-2(a)iii. This gives:

```
...og VP[også beskrevet] NP[de...
```

Word 17 matches rule VP-2a, starting a VP chunk. Word 16 is a noun and matches nothing, so the backwards search stops. This gives:

```
...alternativene] VP[innebærer] for...
```

2.6.3 PP search

The algorithm starts again from the first word, with the sentences chunked with the NP- and VP-chunks as input. The first matching word is word 4, with rule PP-3a. The backward search stops at word 3, since that is a verb, matching nothing. This gives:

```
...VP[har fulgt] PP[opp] NP[dette]...
```

Word 10 matches rule PP-3b partially, but since it is in a chunk already, it do not match rule PP-3b. Word 14 is in the same situation.

Word 18 matches rule PP-3a. Word 17 in the backwards search matches nothing, stopping the search. This gives:

```
...VP[innebærer] PP[for] NP[staten]...
```

Word 25 is in the same situation as word 10 and word 18.

2.6.4 ADJP search

The algorithm starts again from the first word. Word 11, 15 and 22 partially matches rule ADJP-4a, but since they all are in NP-chunks, none matches. It is therefore no ADJP chunks in this sentence

2.6.5 ADVP search

Word 8 partially matches rule ADVP-5a, but since the word is in a VP-chunk, it matches not. No ADVP chunks in this sentence.

2.6.6 Misc search

The algorithm starts again from the first word. Word 7 is the first match, with rule Misc-6b. This rule does not have a backward search.

Rule Misc-6b gives

```
...NP[dette] , CONJP[og] VP[også...
```

Next is word 13, using rule Misc-6a. No backward search.

Rule Misc-6a gives

```
...konsekvensene] SBUP[som] NP[de...
```

Then word 24, with rule Misc-6a. No backward search.

Rule Misc-6b gives

```
...kirke] CONJP[og] NP[andre...
```

Word 27 partially matches rule Misc-6b, but is already in a chunk.

2.6.7 Leftover search

The algorithm starts again from the first word. This search searches all the words, with many partial hits. However, the first hit is word 21. Since the morphology of word 21 marks it as a proper name, and word 22 has word 21 as its head, and word 22's dependency link is of type FLAT, create a new NP chunk for word 22 and word 21. It then checks word 23. Since word 23's head is word 21 and it dependency link is of type FLAT, word 23 is added to the NP chunk of word 21 and 22. This gives:

```
...NP[staten], NP[Den norske kirke] CONJP[og]...
```

The last hit is word 26. Word 26 is an unfinished word, 'ufl', and word 28 is its dependent. Since word 28 is in word 27's chunk, and word 27's chunk is an NP chunk, word 26 is added to the chunk of word 27, according to rule leftover-7(b)iB. In addition, word 25 is a determiner, and word 25's head is word 26. This matches rule leftover-7(b)iB. This gives:

```
...NP[andre tros- og livssynssamfunn].
```

2.6.8 End

This gives us the chunk structure in the start of this section. As can be seen, the NP, VP and PP rules are by far used the most. Some of the rules in the leftovers section match only in a few sentences in the whole corpus.

2.6.9 Implementation

To chunk the Norwegian dependency Treebank using the above rules, we created a program for analyzing the structure using Python. The program is custom made, using only the standard Python libraries. Each rule in 2.5.1 were translated to Python. The rules are run as described above by the program.

2.6.9.1 Output from the program

Each sentence in the results from the program is saved in the same format as used in the gold corpus we are using, that is a format like the ConLL format. This means adding a column to the data. For each word in the dataset, an item designating the chunk is added, the marks are in table 2.3. We are using the same encoding as Sang and Buchholz, see Sang and Buchholz (2000), with *B-<chunk>* marking the start of a chunk, and *I-<chunk>* marking the inside of a chunk. This allows two adjacent words that have the same type of chunk, to be in separate chunks. This is accomplished by the second word having the chunk mark *B-<chunk>*.

We created chunks for all the sentences in the NDT. The used tags are in table 2.3, and the distribution of tags can be seen in table 2.4.

Start chunk	Inside chunk	Notes
B-NP	I-NP	Marks a noun phrase chunk
B-GNP	I-GNP	Marks an owned noun phrase chunk, the owner is last word in the previous chunk
B-VP	I-VP	Marks a verb phrase chunk
B-PP	I-PP	Marks a prepositional phrase chunk
B-SBUP	I-SBUP	Marks a <subjunctional> phrase chunk. This is not used in the Sang and Buchholz (2000)
B-ADJP	I-ADJP	Marks an adjudication phrase chunk
B-ADVP	I-ADVP	Marks an adverbial phrase chunk
B-KONJP	I-KONJP	Mark a conjunction phrase chunk
B-INTERJP	I-INTERJP	Marks a interjunction phrase chunk

Table 2.3: List of chunk marks

Chunk	Amount	Average length
NP	86300	1.50
VP	41984	1.39
PP	40010	1.06
ADJP	10977	1.30
ADVP	5896	1.01
CONJP	7696	1.00
GNP	2365	1.40
INTERJP	223	1.00
SBUP	9857	1.00
0	38673	1.00

Table 2.4: Overview of the chunks placed on the NDT

Part II

Evolutionary Algorithm for predicting chunks using a pre-chunked set

About Part II

A chunker is a program that creates a chunk structure on a sentence. The chunker described in this part shall take as input a sentence tagged with part-of-speech tags, and a set of sentences tagged with part-of-speech and chunk tags. It will then generate chunk tags for the input sentence, using the pre-chunk-tagged set – a fitness set – as a guide. The chunker is a genetic algorithm that uses the training set to calculate fitness using a mathematical formula. This part describes the creating of a GA chunker, and a description and analysis of its accuracy.

Chapter 3

Overview of Evolutionary Algorithms

3.1 Introduction

Evolution is the process with which living beings become better adapted to their environment. Evolution is often described as "survival of the fittest", where fittest can be said to be the being most able to have descendants who are themselves also able to make their own descendants. The process of evolution is most generally the merging of the genes of the parents, and random mutation of genes. In addition to a large amount of luck. The changes enable new beings to be fitter, evolving the population.

Evolutionary algorithms are algorithms that are inspired by the biological process of evolution. They can be used especially for optimization problems, and here we will attempt to use evolutionary algorithms to find the chunks in sentences.

The methods in this and the next chapter are primarily based on the book Eiben and Smith (2007).

3.2 Usage in Natural Language Processing

Evolutionary Algorithms (EA) has been used in NLP, for example for part of speech tagging (Araujo (2002b)) and context-free parsing (Araujo (2002a)). There also exist other works in NLP where an EA has been used.

Araujo used a Genetic Algorithm to predict POS tags on the Brown corpus, using a statistical method in the fitness calculations. This gave the GA accuracy rates comparable to other statistical approaches. The same researcher also used an Evolutionary Programming approach to parsing. Here, a Probabilistic Context Free Grammar was generated to parse sentences, using an EP approach to generate the PCFGs. Results obtained were quite good.

These works shows that while EAs have not been used much for NLP, using an Evolutionary approach can give good results for NLP.

3.3 Introduction to evolutionary algorithms

3.3.1 Representing the problem

An evolutionary algorithm searches for a solution to a problem. To evolve this solution, it needs to be represented in a way that enables the EA to evolve in an effective way. Each such representation of a possible solution to the problem is an individual. An EA works by evolving a population consisting of these individuals.

3.3.2 The evolutionary algorithm

A general evolutionary algorithm follows these steps:

```
INITIALIZE population
EVALUATE each member of population
WHILE stop-criteria not reached DO
    1 SELECT parents
    2 MATE selected parents
    3 MUTATE offspring
    4 EVALUATE offspring
    5 SELECT survivors
SELECT best member as result
```

Initialization EA uses a population that consists of several members. Each member is a representation of a candidate solution of the problem at hand, encoded in a way to enable the functions of EA to work. Before the evolution can start, this population must be initialized.

Evaluation To enable evolution, an algorithm to determine how fit a member is – that is how well a solution solves the problem – must be made. This fitness algorithm is unique to the problem, and must give each member of the population a value that is comparable. All members of the population must have a fitness value attached to enable the EA.

Parent selection During evolution, a selection criterion is used to select members of the population. This selection is the members that are used for mating in the next step

Mating The selections from the previous step are used to create offspring, via a merging of the individuals of each selected member. In biology, two parents are used to create offspring. EAs do not have this restriction.

Mutation Each offspring is randomly changed. Note that the randomness is a central element of mutation, mimicking the biological process.

Survivor selection Another selection criterion is used to select the population used for the next generation. This is usually selected from the combination of the old population plus its offspring or just the offspring. The algorithm then continues evolving the new population.

Termination A stopping criteria, often either time used or an acceptable result achieved, stops the evolution. Here the best member of the population is chosen as the final result from the evolutionary algorithm.

3.3.3 Search space

In optimization, a search space consists of all the possible solutions to a problem. Adding a correctness, or fitness, value to all those solutions, we get an adaptive landscape graph (Eiben and Smith (2007)). This graph can be seen as a landscape where the fitness value of each individual is its altitude. The goal in an EA can be seen to be to move the individuals upward, to get the best fitness score. This landscape consists of tops, where it goes downward on all sides, called local optima. The highest of the tops is the global optimum. The valleys are between the tops, where the fitness values are lower. The goal of an EA is to evolve a population upwards to the tops in the landscape.

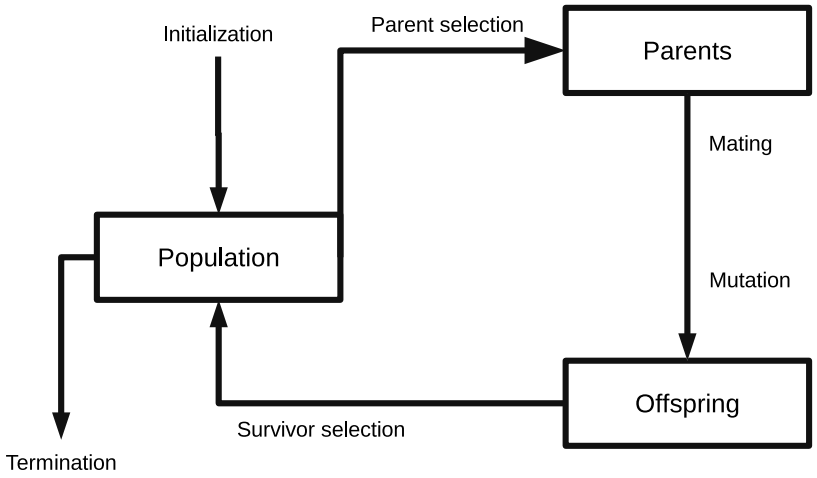


Figure 3.1: General flow of a genetic algorithm

Chapter 4

Predicting chunks using an EA

4.1 Introduction

In this chapter, the specific choices when designing the chunker are presented. The chunker is designed to receive as input a POS tagged sentence, which the chunker will construct chunks for, and a set of POS- and chunk- tagged sentences, which is used to calculate fitness.

4.2 Representation

There are several ways to represent chunks as individuals. This will compare two possible methods.

```
NP[Green(adj) oranges(noun)] VP[flow(verb) badly(adv)]
ADVP[down(adv)] NP[the(noun) river(det)] O[.(punctation)]
```

Figure 4.1: Example sentence used in this explanation

4.2.1 Chunk tag representation

A way of representing chunks is to use the IOB-encoding directly. Each word has an IOB-tag attached, and the list of these tags can be the representation of the chunks in the sentence. An example is in figure 4.2.

```
B-NP I-NP B-VP I-VP B-ADVP B-NP I-NP O
```

Figure 4.2: An individual, the representation of the sentence in figure 4.1. This representation is often used as a starting point in the rest of the text.

This representation allows many clearly invalid chunks to be generated, but the chunker should still be able to generate good results. The representation still have to be of the same length as the underlying sentence, but all the GA operations is much easier to design with this representation. To parse the invalid chunks as correctly made chunks, the representation uses the same procedure for converting invalid lists into valid ones as Ramshaw and Marcus (1995):

- If an I- tag happens without a preceding B- or I- tag of the same type, it is converted into a B- tag. This rule allows errors in the generation to be easily converted into valid representations. Note that this rule is only applied when producing the final result of the GA.

With the length constraint, and the rule about invalid lists above, the initialization of each individual can be designed very easily. The simplest way is to randomly generate a list of chunk tags the same length as the sentence, but many other ways of generating are possible.

The mutation operator is also very easy to design, with a simple random change of a single chunk tag being the easiest.

It is in the design of the crossover operator the advantage of the second representation is clear. The only constraint is to keep a constant length of an individual, so many crossover operators are possible to use. However, when generating chunks for a sentence, the internal order of the chunk tags are important, as a chunk can be over several adjacent words. With this in mind, the crossover operator should attempt to conserve this order. Therefore, the crossover operator can be a simple one-point or multi-point crossover.

4.2.2 Whole chunk representation

Another way is to have a list of items, each with a chunk-type and a length, corresponding to the chunks in a sentence. An example is in figure 4.3.

```
((NP,2)(VP,2)(ADVP,1)(NP,2)(O,1))
```

Figure 4.3: An individual, representing the sentence in figure 4.1

This way of representation ensures that each individual can always be mapped to a possible solution, and no invalid representations can be made. It will also, probably, have a built in tendency for the chunker to create longer chunks. This way of representation does however make the operations of the genetic algorithm much harder to design. An individual must correspond perfectly to the underlying sentence. This means that the individual must be of the same length as the sentence, to avoid either leaving some of the sentence un-chunked, or having chunks that do not correspond to words in the sentence.

The initialization of the sentence must adhere to this constraint, and only create individuals that are of the same and correct length. This can be achieved easily, if slowly, by creating individuals randomly, and throwing away the ones that do not fit the constraint. Alternatively, they can be built a chunk at a time, until the correct length is reached.

The mutation operator can be designed to adhere to this constraint. For the type of the chunk, it can just randomly change it to another type, or swap

the type with another chunk in the individual. For the length of the chunk, it can swap the length with another chunk in the individual. Alternatively, it can swap the position of two chunks in the individual. Changing the actual length of a chunk would mean to also change another chunk with the opposite value to maintain the length of the individual.

```
Initial individual:

((NP,2)(VP,2)(ADVP,1)(NP,2)(0,1))

Random change:

((CONJP,2)(VP,2)(ADVP,1)(NP,2)(0,1))

Length swap:

((NP,2)(VP,1)(ADVP,2)(NP,2)(0,1))

Type swap:

((NP,2)(ADVP,2)(VP,1)(NP,2)(0,1))

Position swap:

((NP,2)(0,1)(ADVP,1)(NP,2)(VP,2))
```

The crossover operation however, is much harder to design than a chunk tag crossover operation. The crossover involves the merging of parts of two or more individuals to create a new individual consisting of parts of all parents. This resulting individual must also be of the correct length, the same as its parents. This can be achieved with choosing one of the parents as the main parent, and crossing over just the type of some of the chunks in the other parents. However, not all parents might have the same number of chunks, so this is also a bit hard to design. Or one of the parents can be chosen as the main parent, and only those chunks in the other parent that are of the same length as one in the main parent are used. But overall, designing a crossover operator in this case is hard.

4.2.3 Used representation

The whole chunk representation is hard to implement, and would, by a high degree of confidence, not give a better final result. It would probably get the result faster, as the fitness function, which is where all the heavy calculations and memory usage are, would spend less time calculating illegal chunk-structures, compared to the chunk tag representation, which creates a lot of illegal chunk-structures. Because the implementation issue, the chunk tag representation were chosen to be our representation, and is used from here on out.

4.3 Initialization

There is a scale on how to initialize the population. The difference is usually if they either give well-enough result in a short time, or very good results in a longer time. One way is to initialize the population in a way that shorten the number of generations that are needed to get a good result, usually by "pre-fitting" the population in some way. This may cause the GA to use less of the possible search space, because the pre-fitting method used do not spread the population so it covers less of the search space, and the GA may therefore give worse results than a more random method. The other way is initializing the population in a way that covers a larger amount of the search space. This may make the GA give better results, but it will spend some more time doing so than the pre-fitted population.

We used the initialization in 4.3.1 in our chunker.

4.3.1 Total random initialization

This initialization is simply choosing chunk tags randomly into a list, until the length of the list reach the length of the sentence. A population of six individuals might look like this:

```
0 I-ADJP B-VP I-CONJP B-CONJP B-ADJP 0 I-NP
I-ADJP 0 B-INTERJP I-VP B-INTERJP B-NP B-INTERJP B-CONJP
B-NP I-NP I-ADJP 0 0 B-NP B-NP 0
0 I-NP B-ADJP I-VP I-NP B-NP B-NP B-CONJP
B-NP I-CONJP 0 I-VP B-INTERJP B-NP I-NP 0
0 B-INTERJP 0 I-VP I-CONJP B-NP I-NP B-INTERJP
```

Neither of these are of course not a remotely sensible chunk tagging of sentence 4.1. Sentences almost never start with 0, and INTERJP is very rare. There are also lots of I- tags without a corresponding B- tag. However, a large population should cover the whole search space quite good. This will enable the EA to, compared to a more fit initialization, evolve more easily towards the global optimum, if this optimum is far from the center of the more fit initialization.

4.3.2 Random initialization plus added most-common chunk tag lists

This initialization consists of some part of the population generated with the method described in 4.3.1. The rest of the population is created by selecting the most common chunk tag for the part of speech tag in the evaluation set for each word in the sentence. The resulting population might look like this, with three individuals randomly generated:

```
B-NP I-NP B-VP I-VP B-ADVP B-NP I-NP 0
B-NP I-NP B-VP I-VP B-ADVP B-NP I-NP 0
B-NP I-NP B-VP I-VP B-ADVP B-NP I-NP 0
0 I-NP B-ADJP I-VP I-NP B-NP B-NP B-CONJP
B-NP I-CONJP 0 I-VP B-INTERJP B-NP I-NP 0
0 B-INTERJP 0 I-VP I-CONJP B-NP I-NP B-INTERJP
```

A population of 6 individuals, representing the sentence in figure 4.1.The three first are most-common chunk, the other three random initialized.

The randomly generated individuals have the same problems as in 4.3.1. The individuals generated by selecting the most common chunk tag are starting much farther up a slope in the landscape graph, and will have to change far less to become a good result. However, they might not start at the slope giving the best result, and will therefore need to jump valleys to get to the best slope to have the best result.

This initialization will, depending on the percentages of each type of method, still cover the search space. However, the individuals that are "pre-fitted" will be closer to a correct solution, and will spread their contents with the others during crossover, and thereby hinder the GA from searching a larger part of the search space. A good result should be generated somewhat quicker using this method however, as fewer generations is required to get a good result.

The implementation has the same constraint regarding possible tags as 4.3.1.

Variants A simple variant of this is to mutate each of the pre-fitted individuals, to avoid having a large amount of identical individuals.

4.3.3 Other initialization methods

There exists other ways to initialize our populations, and all are different ways of pre-fitting the population. The most interesting for our purposes would be to select a chunk tag from a list of the possible chunk tags as exist in the training set for that word's part-of-speech tag. This can be done either completely randomly, or according to the distribution of chunk tags for that word's part-of-speech tags.

4.4 Mutation

There are also many ways to mutate an individual. The idea of mutation is to introduce a small change in the individual randomly, mimicking the biological concept of mutation. This small mutation is meant to move the individual slightly in the search space, perhaps making it more fit. Note that the idea behind mutation is to make random changes, without regard for making a sane chunk structure.

4.4.1 Random resetting of chunk-tags in a individual

Our representation – a list of chunk-tags – can be seen as an integer representation, where each chunk-tag corresponds to a unique integer value. We can therefore use a method used on integer representations (see Eiben and Smith (2007, p.43)), with the restriction that only the integers that correspond to a chunk tag are allowed:

```
random_reset(individual):
  for each chunk-tag in individual:
    if random_float(0,1) < mutation-rate:
      new-tag := random choice drawn from pool
        replace chunk-tag with new-tag
```

Running this method with our example sentence 4.1, could for example give:

```
random_reset(B-NP I-NP B-VP I-VP B-ADVP B-NP I-NP O) =>
  I-ADJP I-NP B-VP I-VP B-ADVP B-NP I-NP O
First chunk tag is changed with I-ADJP
```

4.4.2 Scramble a subset of an individual

This method is based on the method from the permutation type representation (see Eiben and Smith (2007, p. 46)), where the mutation-rate is the chance of each individual being mutated, instead of each chunk tag in each individual. This method randomly draws two positions in the individual, and draws new chunk tags from the pool of possible chunk tags for all the genes between those two positions, in essence re-initializing part of the individual.

```
scramble(individual):
  r1 <- random_int(0, length of individual)
  r2 <- random_int(0, length of individual)
  if(r2 > r1): r1,r2 <- r2,r1
  for each chunk-tag in individual from r1 to r2:
    new-tag <- random choice drawn from pool
    replace chunk-tag with new-tag
```

Here, the mutation rate is the chance of each individual to be mutated, and this can be anything from 0.0% to 100%.

Running this method on our example sentence 4.1, would for example give:

```
scramble(B-NP I-NP B-VP I-VP B-ADVP B-NP I-NP O) =>
  B-NP I-ADVP O I-VP B-CONJP B-NP I-NP O
r1=2,r2=5, I-ADVP O I-VP B-CONJP is inserted
```

4.4.3 Comparison of the two methods

Method 4.4.1 and 4.4.2 have different effects. Method 4.4.1 will on average only change the number chosen as average chunk tags in each individual, but it might also change many more chunks, depending on the random result. In the case where it changes many chunk tags, it might destroy the order in the individual, making that individual much less fit. Method 4.4.2 on the other hand, can also scramble a whole individual, but when it scrambles only part of an individual, the rest of the individual keeps the order intact. The change the mutation operator does is in effect much harder to control using 4.4.2 than using 4.4.1. This could of course be alleviated by changing the algorithm of 4.4.2 somewhat, but for our use, we choose the method in 4.4.1.

4.5 Crossover

In crossover, we are also restricted to only some of the possible methods. The idea of crossover is to blend the representations of two or more parents, creating a new individual offspring sharing some aspects from all its parents. This of course mimics the biological process of reproduction, although that process has in nearly all cases just two parents.

In our crossover operators, we are restricted to only those that can be applied to our representation, and that representation can be considered as an integer representation using integers from a limited pool. However, our representation also has important information encoded in the order of each chunk tag, namely the complete chunks. This means that our crossover operators should attempt to preserve some of this order in each parent, while still blending the genes of each parent to create a new offspring.

4.5.1 One-point crossover

This is an adaption of the crossover described in Eiben and Smith (see Eiben and Smith (2007, p.47)). This is a very simple method creating two offspring from two parents. It works by simply selecting a position at random, then taking all chunk tags to the left of this point from parent 1, and combining this with all tags to the right of parent two. The second offspring is built by taking the leftovers from this operation.

```
one-point(parent1,parent2):
  position <- random-integer(0, length(parent1))
  offspring1 <- parent1[0 : position] +
                parent2[position : end-of-individual]
  offspring2 <- parent2[0 : position] +
                parent1[position : end-of-individual]
```

The input to this method is both parents who are creating an offspring, and the output is two offspring. Running this method will for example give:

```
one-point(parent1=[B-NP,I-NP,I-NP,B-VP,B-ADJP],
          parent2=[B-ADJP,I-PP,B-CONJP,B-NP,I-ADVP])
  position <- 3
  offspring1 <- [B-NP,I-NP,I-NP] + [B-NP,I-ADVP]
  offspring2 <- [B-ADJP,I-PP,B-CONJP] + [B-VP,B-ADJP]
```

This method is very simple, and it preserves the order in the parts from each parent well. It is the crossover operator used in the GA chunker.

Variant Very simple variants of this can be made by adding more points, as in "two-point crossover" or "three-point crossover".

4.6 Evaluation

The evaluation method is the method that evaluates each individual, and gives each individual a score, which is used to compare the individuals to each other. The method should give scores in such a way that the individual that solves the problem most correctly should have the highest score, while the worst individual should have the lowest. It is important to note that the numerical value of the scores are not important, the only thing of importance are that the method should be able to rank the whole population according to how each individual solves the problem.

For a simple problem, like the Traveling Salesman Problem, or the Knapsack problem, the evaluation function is simple to write. For the TSP, it is simply the sum of the distances between the cities, according to the order specified in the individual being evaluated. The lowest distance gets the highest score, making it more likely to be mated and creating offspring.

Our chunking problem does not have such a simple evaluation of possible solutions. We instead have to use a training set, consisting of already chunked sentences, to generate a score for our individuals.

4.6.1 Fitness score of a single individual

To score an individual we need to calculate its correctness according to the training set. This correctness, or fitness, is then used to rank the individuals in the population. The calculation depends on extracting features from the evaluation set. Note that the rule for converting an invalid chunk structure into a valid one, described in 4.2.1, is not applied before calculating fitness.

4.6.1.1 Features used

In the individual that are being evaluated, the following data is available to the fitness calculation:

- The word itself
- Part of speech of each word
- Other information that can be constructed from those two
- The genes in the individual, namely the chunk tags.

There are many ways to create a ranking between different individuals from this information. In the formula that was created, a focus on the long, complete chunks was used. This means that the formula uses the count of increasingly long strings of chunks, up to 5 in either direction, to generate a fitness score. The features used was inspired by the features used in the work by Araujo (2002b) and the works in the ConLL2000 task.

Only the POS of the sentence and the chunk tags in the individual were used, and not the words themselves. This was done for performance reasons, as the data tables would be very large and slow when using words.

4.6.1.2 The formula

The calculation follows these steps:

1. For each chunk tag y in the individual, a fitness of that chunk tag is computed according to the formula

$$g(y) = \sum_{i=1}^{11} f_i(x) \tag{4.1}$$

Where f_i is a function i, y is the chunk tag and x is the position of the chunk tag in the individual. The function f_i is defined as

$$f_i(x) = \begin{cases} ln(m_i) + ln(c_i(x)) & if \quad c_i(x) > 0 \wedge m_i > 0 \\ s & if \quad c_i(x) = 0 \wedge m_i > 0 \\ 0 & if \quad c_i(x) = \text{undef.} \\ 0 & if \quad m_i \leq 0 \end{cases} \tag{4.2}$$

The features in c_i is defined as

$$c_1(x) = occ(t_x, c_x) \tag{4.3}$$

$$c_2(x) = occ(t_{x-1}, t_x, c_{x-1}, c_x) \tag{4.4}$$

$$c_3(x) = occ(t_{x-2}, t_{x-1}, t_x, c_{x-2}, c_{x-1}, c_x) \tag{4.5}$$

$$c_4(x) = occ(t_{x-3}, t_{x-2}, t_{x-1}, t_x, c_{x-3}, c_{x-2}, c_{x-1}, c_x) \tag{4.6}$$

$$c_5(x) = occ(t_{x-4}, t_{x-3}, t_{x-2}, t_{x-1}, t_x, c_{x-4}, c_{x-3}, c_{x-2}, c_{x-1}, c_x) \tag{4.7}$$

$$c_6(x) = occ(t_x, t_{x+1}, c_x, c_{x+1}) \tag{4.8}$$

$$c_7(x) = occ(t_x, t_{x+1}, t_{x+2}, c_x, c_{x+1}, c_{x+2}) \tag{4.9}$$

$$c_8(x) = occ(t_x, t_{x+1}, t_{x+2}, t_{x+3}, c_x, c_{x+1}, c_{x+2}, c_{x+3}) \tag{4.10}$$

$$c_9(x) = occ(t_x, t_{x+1}, t_{x+2}, t_{x+3}, t_{x+4}, c_x, c_{x+1}, c_{x+2}, c_{x+3}, c_{x+4}) \tag{4.11}$$

$$c_{10}(x) = occ(t_{x-1}, t_x, t_{x+1}, c_{x-1}, c_x, c_{x+1}) \tag{4.12}$$

$$c_{11}(x) = occ(t_{x-2}, t_{x-1}, t_x, t_{x+1}, t_{x+2}, c_{x-2}, c_{x-1}, c_x, c_{x+1}, c_{x+2}) \tag{4.13}$$

$occ(n)$ is a function that gets the occurrences of n in the evaluation set. See table 4.1 for an example of a table. t_x is the part of speech tag of the word at position x in the sentence the chunker is chunking. c_x is the chunk tag at position x in the individual the chunker is evaluating. m_i is a number that is used to smooth each function, see 4.6.1.3, and if $m_i <= 0$, $f_i = 0$. If the position x is such that some part of the feature is outside the bounds of the individual – for example at $x = 2$, when t_{x-4} will be the part of speech type at position -2, which do not make sense – $c_i(x)$ is undefined and that feature will not be used.

2. In addition for the function $occ(x)$, if x does not have an entry in the evaluation set, a number $s = ln(w/length(sentence))$, where w is a number, the same in all c_i, is returned instead.

3. The total sum for an individual is calculated according to the formula:

$$Total fitness = \sum_{x=1}^{n} g(x) \tag{4.14}$$

Where $g(x)$ is function 4.1

The total from function 4.14 is the fitness of the individual. This number is used by the parent and survivor selection step to select which parents should procreate and which member of the population should be replaced. It is also used to select which individual is the final result of the chunker at the end of the GA.

Type_1	Type_2	Type_3	Chunk_1	Chunk_2	Chunk_3	Count
adj	adj	subst	NP-B	NP-I	NP-I	1123
adj	adj	subst	NP-I	NP-I	NP-I	523
adv	adv	verb	VP-I	VP-I	VP-I	112
<kom>	verb	subst	0	VP-B	NP-B	352

Table 4.1: A part from the set using to calculate fitness. Table for function occ in f_3.

4.6.1.3 Analysis

The numbers m_i in each function is a smoothing of the occurrences in the data set. In essence, if $occ(n) = 1$, then f_i will become $f_i(x) = ln(m_i) + ln(1) = ln(m_i)$. This essentially multiplies all occurrences in the dataset by m_i. Additional increases of $occ(n)$ will only add $ln(occ(n))$ to the score.

Our fitness formula will therefore give the best score to a chunk tag in an individual, when compared to the other possible chunk tags:

- Gets at least 1 hit in the dataset for each function $c_i(x)$. This adds all m_i to the fitness value.
- Gets the maximum total number of occurrences combined, across all the different $c_i(x)$.

The best score for an individual in the population will then be one that:

- Gets at least 1 hit in the dataset in each function for each chunk tag.
- Gets the maximum total number of occurrences combined across all the functions for all the chunk tags.

4.6.2 Why not a probability function

Instead of using the formulas above for calculating the fitness of a gene, we could have calculated the fitness of that gene using formulas on the pattern of:

$$f_{3_x}(x) = log(m_{3_X} \cdot \frac{occ(t_{x-2}, t_{x-1}, t_x, c_{x-2}, c_{x-1}, c_x)}{occ(t_{x-2}, t_{x-1}, t_x)})$$

This calculates the probability of a string of chunk-tags given a string of POS-tags, then multiplies that probability with a weight, then takes the logarithm of the result. These were not used for a reason:

We are not after the probability for a gene directly, just how well it fits according to the training-table. For example, using the table 4.1 as a training table, with $m_{3_x} = m_3 = 1$:

$$f_{3_x}(adj, adj, subst, NP-B, NP-I, NP-I) = log(1 \cdot \frac{1123}{1646}) = -0.3823$$

$$f_{3_x}(adj, adj, subst, NP-I, NP-I, NP-I) = log(1 \cdot \frac{523}{1646}) = -1.1465$$

$$f_3(adj, adj, subst, NP-B, NP-I, NP-I) = log(1 \cdot 1123) = 7.0237$$

$$f_3(adj, adj, subst, NP-I, NP-I, NP-I) = log(1 \cdot 523) = 6.2595$$

$$-0.3823 - -1.1465 = 0.7642 = 7.0237 - 6.2595$$

As can be seen, function f_{3_x}gives lower values than function f_3. However, the difference between the two is the same, as can be seen above. Since we are only interested in whether one individual is more fit than another, this is all the matters. Not using probabilities effectively halves the number of lookups the evaluating function have to do, which saves time, but gives the same result.

4.7 Parent selection and Population Model

We are using a steady-state population model (Eiben and Smith (2007, p. 58)) for our GA. In this model, λ offspring are created from a population of μ members, so that each generation λ members of the population is replaced by the generated offspring. In our case, we choose $\lambda = 1$, so that each generation 1 offspring is created, and 1 member of the population is replaced with that offspring. As a consequence of only 1 new member of the population being created each generation, resulting in very many generations.

4.7.1 Tournament selection

Tournament selection is a simple method to draw parents from the population pool. It works by drawing n parents from the pool randomly, then ranking the drawn parents according to their fitness score. The individual with the best fitness score is added to the pool of parents that are used for creating offspring.

```
tournament_selection(population,n):
   tournament <- {}
   loop n times:
      r <- random_integer(0,size of population)
      tournament <- tournament + population[r]
   sort(tournament,fitness score)
   return tournament[0]
```

In GA's it is important to have a good selection pressure (Eiben and Smith (2007)). The selection pressure is a pressure to select the best of the population for mating, to enable the GA to evolve the population upwards in the search space. But it is important to also preserve some of the less fit individuals in a population, as without them the GA might get stuck on a local optimum in the search space. In tournament selection this pressure is easy to control. If $n = size(population)$ only the best fit member of the population will be selected for mating, while if $n = 1$, the draw will be random, and the selection pressure will be zero. A n between these two extremes is usually good, and

should typically be related to the population size. Tournament selection also does not require knowledge of the whole population, and this makes it good to be used if a distributed system is required. Tournament selection is also quite easy to implement, and also cheap computationally. Note that in this method, the same member can be selected several times.

4.7.2 Survivor selection

Here we are selecting which of the members of the population is replaced with the offspring created by crossover and mutation. This selection is a reverse tournament selection as described in 4.7.1, where we select the worst member of a group of n members of the population to be replaced by the offspring. This reverse tournament have the same features as described above, and is just as easy to implement and to be used in a distributed system. It is also an elitist system, the best member of the population has no chance of being replaced by a offspring.

4.8 Stopping the GA

We need to stop the GA at some point. Since we cannot say when the fitness is as good as it can be, some other things can be used. This can be many things; number of generations done, actual time used, the difference between members in the population being low, and some others.

We keep a running count of the number of generations since the best-fit individual changed. This number starts at 0, and increases by 1 for each generation, unless a better-fit individual than the best-fit individual is created. In that case, the number is reset to 0. If this number reaches $G*length(sentence)$, the GA stops, and the best-fit individual is produced as a result from the GA. This ensures that the GA does not stop when it is actively producing better individuals, and stops when it have spent too much time to produce a better individual. Since longer sentences have larger search spaces, and will take longer to create a better member, we adjust the stopping number according to the length of the sentence being chunked. This number, G, can easily be adjusted depending on how good results are wanted, at the cost of time.

4.9 Implementation of the GA chunker

Our GA is implemented in Java, using the DEAP system (Fortin et al., 2012). Python were attempted at first, but Python proved to be too slow to be useful, and the reimplementation in Java were at least a factor of 10 faster. Since DEAP is a Python package, the parts we used of DEAP was reimplemented in Java. No other packages other than the Java standard library are used. Below are details about each section of the implementation.

4.9.1 Startup and data structures

At startup, the set used for fitness computing is read into memory. To increase the speed of the chunker, each word, chunk tag and POS tag are mapped to integers, as integers are much faster to work with than strings.

Initialization of evaluation data The evaluation data are generated from the training set, and consists of tables like the table 4.1, one table for each function. We use one table for each function to avoid overlap. Since these tables would be very large when implemented as arrays – the largest function, function f_{11}, would use a 10 dimensional array – , we choose to instead implement the tables using Java's HashMap function, as this function is reasonably fast, and also space efficient. The keys to the HashMap are generated with the functions:

$$f(x, y) = x * c^y \tag{4.15}$$

$$Key(x_1, x_2, ..., x_n) = f(x_1, 0) + f(x_2, 1) + ... + f(x_n, n-1) \tag{4.16}$$

Where x_n is the integer mapped POS-tag or chunk-tag, n is the position of the POS-tag or chunk-tag, and $c-1$ is equal to the largest valued POS or chunk integer. This is essentially a positional base-c system. For example, the key for $occ(t_{x-1}, t_x, c_{x-1}, c_x)$ in function f_2, using the sequence of POS- and chunk-tags:

```
adj (10), subj (2), NP-B (2), NP-I (13)
```

And a $c = 20$, would be:

$$Key(adj, subj, NP-B, NP-I) = 10*20^0 + 2*20^1 + 2*20^2 + 13*20^3 = 104850 \tag{4.17}$$

While the speed of which we create the training tables are not important, as that only happens once at startup, these functions are also used when generating keys in the evaluation of an individual, which happens once at initialization of the population for each individual, and once each generation for the offspring.

The hashing function we are using, the one described above, can create overlapping hashes when we are using functions with the same number of inputs. Functions f_3, f_7 and f_{10} all have the 6 inputs and functions f_5, f_9 and f_{11} have 11 inputs. We are therefore using one training table for each function. This does not have an effect on the memory usage of the GA, since all it adds is a tiny bit of overhead.

4.9.2 The Genetic Algorithm

The main GA algorithm generates a chunk structure according to the methods described in section 4.3-4.7. In the implementation, it gets a single POS tagged sentence, together with the training set, and the algorithm then generates a chunk structure for that sentence.

Representation of the solutions Each possible solution in the population is represented using an array of integers, and the crossover and mutation operators work on this array. In addition, each individual has a fitness value.

Initialization of the population The population is implemented as an array of individual arrays. Each individual in the population is initialized according to the method in section 4.3, using an integer mapped representation. Each individual is then evaluated for fitness, so that the GA can start.

Main loop Having initialized the population, the GA then runs the loop described in section 3.3.2.

1. The tournament selection with k participants is run once, to select the primary parent

2. A random roll is done, to determine if a crossover is to happen this round

   (a) If a crossover is to happen, a second tournament selection is done, using the same k as the main one, to select the secondary parent. The primary and secondary parents are then used in the algorithm described in 4.5.1. Only the first offspring from that algorithm is used, and becomes the offspring for this generation. Since the offspring might be different from its primary parent, it needs to be evaluated for fitness. A flag is therefore placed on the offspring, marking it for reevaluation.

   (b) If a crossover is not to happen, the offspring is created asexually, as a clone of the parent selected in 1). Since it is identical to its parent, no flag is placed on it. This is done to give the GA a chance of mutating an existing individual, without the mutated individual being the product of crossover.

3. The mutation operator described in 4.4 is then applied to the offspring. If any of the genes in the individual are mutated, a flag is placed on it, marking it for reevaluation.

4. The evaluator is run for the offspring, if it has the flag set. This calculates the fitness value of the offspring. The flag is used because the evaluator is the most costly operation the GA uses, by several orders of magnitude. If an offspring do not need to be reevaluated, the GA can save time by not evaluating them.

5. For survivor selection, a reverse tournament with the same k as the main one, selecting the individual to be replaced. This individual's genes are overwritten by the offspring's genes, and its fitness value is overwritten with the offspring's fitness value.

End After the main loop is finished, according to the criterion in section 4.8, the GA returns the best-fit member of the population, after translating it into proper chunks. To translate it into proper chunks, the rule described in 4.2.1 is used. Not that this is the only time that rule is used in our GA.

As checking for the best-fit member of the population involves sorting the whole population, we only check the stopping criteria every thousand generation.

Chapter 5

Results from the Evolutionary Algorithm

5.1 Introduction

In this chapter is the result from the chunker created in chapter 4, and some analyses of some of the variables used in the chunker. Also presented are some methods that can improve the chunker.

5.2 Experimental setup

As seen in chapter 4, the chunker has many parameters to tweak – mutation and crossover rate, size of the population, size of the tournament, and all the weighting for the evaluation formula in 4.6.1. These parameters need to be set at certain values to get the best possible result. Since each GA is unique, these parameters must be determined experimentally. Another GA were used to determine these parameters.

5.2.1 The GA used to determine the best parameters to use in the main GA

For determining the best parameters, a rather simple GA was used. For this, DEAP (Fortin et al., 2012) was used, the same as in the main GA, but it was not converted to Java. The best variable determiner GA (BVDGA) simply ran the main GA and tested the output of different sets of parameters.

To reduce the amount of time used, the task of determining the best parameters were divided in two. Before start, the crossover, mutation, population size, and tournament size parameters were hand-tested, until good enough parameters were determined. Then the weights of the evaluation formula were determined by the BVDGA. After that was done, the crossover, mutation, population size, and tournament size were determined by the BVDGA. The BVDGA was therefore used twice to determine the best parameters. This might have introduced some error, as some errors might have been introduced by determining the weights and the rest of the parameters separately. However, the weights

and the rest of the parameters should not depend to heavily on each other.

Hardware used For running these experiments, a Dell PowerEdge M910 was used, equipped with 4 Xeon L7555 at 1.87 GHz, making for 32 physical cores and 64 total virtual cores with hyper-threading, and 128 GB ram, running Windows Server 2008 R2.

5.2.1.1 Data set used for determining parameters

When determining the parameters, we used the chunks generated by our chunking program detailed in section 2.6.9. For input to that program, we chunked version 0.3 of the Norwegian dependency Treebank (*Norsk dependenstrebank* 2014), except 110 sentences that were kept away for usage as a validation set. This gave **9870** chunked sentences of varying length. Most of the sentences were from newspaper articles, this gives the sentences a longer than average length than normal Norwegian sentences.

This set of **9870** sentences were then grouped according to sentence length, and **2%** of each length group were put into the set used for testing the different parameters, the test set, while the rest **98%** of the sentences were used for the set that evaluates each individual in the GA, the training or fitness set. This gave us a test set consisting of **239** sentences, and a fitness set consisting of **9541** sentences. As a consequence of the way the test set was created, it consists of sentences with lengths spread across the whole spectrum of lengths. Structuring the test set should give us a test set that better represents all the different parts of the Treebank than the alternatives. The alternatives is either a random draw, which can end up missing some parts of the Treebank, or a selected set of sentences, which need to be constructed carefully to get a representative selection.

5.2.1.2 Description of the BVDGA for determining the weights to the fitness formula

This was done in several stages. Since this part took many months to calculate, we ran the BVDGA several times, using the results from the previous run as an initializer for the next run. This was done partially to help manually guide the process, and partially because of hardware failure on the server we used. The manual guiding we did were to change the rate of the change in mutation, from very large at the start, to low at the end. We chose to end the generation when we felt good enough results had been achieved.

Since we here were looking for the weight parameters only, the crossover, mutation, population size and tournament size for the main GA were locked too:

- Mutation = 5%
- Crossover = 45%
- Population size = 1000
- Tournament size = 4
- GA termination criteria (see 4.8)= 500*sentence_length

Representation Here, the representation was easy to do, as all the parameters are simply numbers. So the representation was the numbers themselves, we used simply a list of 12 floating point numbers, representing each of the weights in 4.6.1, and the last representing the empty weight.

Initialization Since we did not have any indications of a good starting position for the weights, each gene were initialized to a random number between 1 and 10000000. After having gotten an indication of where the numbers should be, the algorithm was restated using the input from the previous run.

Mutation The mutation operator was based on a variant of the method "Nonuniform Mutation with a Fixed Distribution" described in Eiben and Smith (2007, p. 44). This method works by adding to each gene a random value drawn from a standard distribution with a mean of **0** and a standard deviation. The standard deviation chosen varied, from a **100** times the current value of the gene at the first run, to $\frac{1}{5}$ of the current gene value at the last run. Instead of applying this mutation to all the genes, as in the book, we used a chance of **30%** at each gene to determine if it should be mutated. This was to lower the rate of change, and stabilize the outputs a bit more than would otherwise happen. In addition, if a gene value went below **0**, it was raised to **0**, to avoid having negative weights.

Crossover For crossover, we used a one-point crossover identical to the one described in 4.5.1. The crossover chance was at **40%**.

Evaluation and controlling for variance The evaluation here is simple. Each individual is used to run the main GA, and the $F_{\beta=1}$ score produced is used as our fitness. Since the result from the main GA varies depending on the random input to the GA (see section 5.5.1 for an analysis of that), we attempted to control for that variance. This was done by running the main GA **20** times, and calcuating the mean $F_{\beta=1}$ score of these results.

Parent selection and population model Here, we used a generational model (see Eiben and Smith (2007, p. 58)). From a population of size ţ, λ offspring is created, where $\lambda = ţ$. Then these offspring becomes the next generation's population. This means that the survivor step is skipped, as all individuals perish. In addition, elitism is used, the most fit member of the current population is always kept and brought over to the new generation; this also replaces one of the offspring.

To fill the mating pool, a tournament selection was held, once for each member of the population. This tournament selection is identical to the one described in 4.7.1, using a tournament size of **4**.

Fitness Since the result from the GA-chunker varies depending on the random seed of the chunker (see section 5.5.1 for an analysis of that), we attempted to control for that variance. This was done by running the chunker **20** times, using the same parameters but with different seeds[1], then taking the average of the resulting $F_{\beta=1}$ scores as the fitness value of the individual in the BVDGA.

[1] Seed values at 65 to 85, not that this matters

Results The final weights our BVDGA determined gave best result are in table 5.1. As can be seen, several of the formula weights are at **0**, which means that these functions to these weights were determined to have a negative effect on the final result – if they had no effect they would just be ignored – and therefore the weights is at zero.

f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}
0	115	0	10	53	0	17	76	0	0	484208194	1

Table 5.1: Values from the BVDGA

5.2.1.3 Description of the BVDGA for determining the evolutionary parameters to the GA chunker

Unlike the BVDGA determining the weights, this was done in a single run, since the parameters were far faster to determine, as they where fewer. In addition, we had a slightly better idea of were the parameters should be at the start, from testing done during development. At the start we had these parameters:

- Population size = 400 individuals
- Tournament size = 15
- Mutation rate = 25%
- Crossover chance = 45%
- GA termination criteria = 250*sentence_length

As can be seen, compared to the final output from the BVDGA, we started too high on all parameters except the population size.

Representation The representation here was simply that population size and tournament size are integers, and mutation and crossover rate are floats limited to between 0.0 and 1.0.

Initialization The parameters were initialized to the values described above, but each value was also changed by a number drawn from a randomly from a Gaussian distribution with a mean of **0** and a standard deviation of $\frac{1}{5}$ of the value of the gene.

Mutation This used the exact same method as the mutation in 5.2.1.2, but with a standard deviation at $\frac{1}{5}$.

Crossover For crossover, we used a one-point crossover identical to the one described in 4.5.1. The crossover chance was at **40%**.

Parent selection and population model This used the same population model as the population model in 5.2.1.2.

Fitness The fitness was calculated using the same method as in 5.2.1.2.

Result The final parameters our BVDGA determined gave best result are in table 5.2. As can be seen, the crossover rate is very low. This is analyzed in section 5.4, but as a short description, at a low level of mutation rate, the crossover rate have a very small, but negative, effect on the final result.

Mutationrate	0.042
Crossoverrate	0.001
Populationsize	1678
Tournamentsize	10

Table 5.2: Values from the BVDGA

5.3 Effectiveness of the chunker on the Norwegian Dependency Treebank

5.3.1 Test setup

To evaluate the effectiveness of our GA chunker we are using 9-fold[2] cross validation on version 1.0 of the NDT, which have been chunked by the rule set in part I. This gives us 20045 sentences, and each fold consists of 2226 sentences in the validation set, and 17817 in the fitness set. Our parameters were the ones given by the BVDGA in 5.2.1, detailed in table 5.2 and 5.1.

We are using F-score as our main measure, with precision and recall as side measures. The F-score is defined as $F_\beta = (1 + \beta^2) * \frac{Precision*Recall}{\beta^2*Precision+Recall}$, giving $F_1 = 2 * \frac{Precision*Recall}{Precision+Recall}$. When we measure the precision and recall, the criteria for a hit is that the whole chunk matches. The type of the chunk must be the same, and it must start and stop at the same positions in the sentence. Anything else is considered a miss. We do not count O chunks at all. An O chunk – note that all O chunks are single word chunks – in the generated sequence of chunks is not a hit or a miss in both precision and recall. An O chunk in the sentences we are measuring against need therefore not be matched in the chunk sequence the chunker have generated. The rationale here is that O chunks are very easy to predict in our set, as they are nearly always punctuation marks. Note that if our chunker has placed a not-O chunk in over a word that is an O chunk, a precision miss is counted.

Each category of chunk is counted separately, while the ALL category is all the chunks added together. The ALL category is not the average of the other classes, but calculated by adding all the hits and misses from all the other chunk classes. This should be the same methods of counting precision and recall as in the ConLL2000 task, however, we can not find their definition of what constitutes a hit or miss. But we can observe that our baseline in chapter 6 is identical to the one in the ConLL2000 task.

We are also presenting the average length of the chunks our chunker produces. This can be compared to the output from part I, where the average length of chunks is shown in table 2.4. The average length shown below is only counting the chunks the chunker managed to predict correctly according to the verification set.

[2] 9 because we could run 9 GAs simultaneously on our server

We are running each fold separately, and then averaging the results from those runs. We are then running each validation **20** times with different seeds, then averaging those results, making the final results the average of the average of each fold. This is to reduce the variance of the results, and presenting a more correct assessment of the effectiveness of the chunker. The chunker is quite slow; The largest stop value used using about 90 hours to complete one 9-fold crossvalidation.

Note that the sentences we are chunking using our chunker are POS tagged to a gold standard, and we expect to get better results than using an automatic POS tagged corpus.

5.3.2 Baseline

As a baseline, we placed the most common chunk tag on each part of speech tag. We present the results using the same folds as used in the cross-validation.

Chunk type	$F_{\beta=1}$	Precision	Recall	Average length of correct chunks
NP	0.570	0.497	0.669	1.0
VP	0.587	0.518	0.677	1.0
PP	0.923	0.908	0.940	1.0
ADJP	0.424	0.301	0.725	1.0
ADVP	0.647	0.483	0.981	1.0
CONJP	0.835	0.719	1.000	1.0
INTERJP	1.000	1.000	1.000	1.0
SBUP	0.999	0.999	1.000	1.0
GNP	0.000	0.000	0.000	NA
ALL	0.650	0.571	0.756	1.0

Table 5.3: Baseline result of English set

As we can see in table 5.3, the baseline does manage to predict some chunks, with an average $F_{\beta=1}$ score for all chunks of **0.65**. However, all the chunks it manages to predict are single word chunks. The minimum average length of a correct chunk will always be **1.0** as that is the shortest a chunk can be.

INTERJP and **SBUP** chunks are noteworthy, as the baseline places these with perfect precision and recall[3]. This is because both of these are always single word chunks, and that the POS tags of type **interj** and **sbup** is always an **INTERJP** or **SBUP** chunk. This can be compared to the **CONJP** chunks, which have a perfect recall, but lower precision. This is because a **konj** POS can either be part of a single word **CONJP** chunk, or be part of a longer **NP** chunk. As **konjs** is most often part of a **CONJP** chunk, the baseline will always assign a **konj** to a **CONJP**, and thereby achieve perfect recall.

We must also note the **GNP** chunks, which have an F1 score of **0**. This is expected, as the POS tags contains no information about them. The length of the GNP chunks is **NA**, as the baseline did not manage to predict a single chunk.

[3]The .999 score on SBUP can be attributed to errors in the NDT

5.3.3 Results at stop criteria at $S = 250$, $S = 2000$ and $S = 16000$

Chunk type	$F_{\beta=1}$	Precision	Recall	Average length of correct chunks
NP	0.849	0.845	0.853	1.40
VP	0.953	0.951	0.955	1.36
PP	0.924	0.922	0.925	1.03
ADJP	0.759	0.719	0.806	1.19
ADVP	0.736	0.652	0.844	1.00
CONJP	0.860	0.832	0.889	1.00
INTERJP	0.760	0.647	0.955	1.00
SBUP	0.991	0.989	0.993	1.00
GNP	0.139	0.214	0.104	1.60
ALL	0.877	0.869	0.885	1.26

Table 5.4: Results at $S = 250$

Chunk type	$F_{\beta=1}$	Precision	Recall	Average length of correct chunks
NP	0.861	0.856	0.866	1.41
VP	0.961	0.959	0.964	1.36
PP	0.930	0.928	0.932	1.03
ADJP	0.781	0.745	0.821	1.19
ADVP	0.756	0.676	0.858	1.00
CONJP	0.873	0.847	0.901	1.00
INTERJP	0.907	0.844	0.990	1.00
SBUP	0.997	0.996	0.998	1.00
GNP	0.149	0.278	0.103	1.59
ALL	0.889	0.882	0.895	1.26

Table 5.5: Results at $S = 2000$

As we see, the results get better at higher stopping criteria, as is expected. We do however note:

- PP chunks increased only very marginally compared to the baseline, and each other. Recall even sank at $S = 250$ and $S = 2000$ compared to the baseline. Since the F-Score is so high, further increases at higher stop values will be low.
- NP chunks show a clear increase from stop value to stop value, and the average length of a NP chunk also increases towards the correct average. The NP chunks should continue to improve at higher stop criteria.
- At $S = 16000$, VP chunks are almost perfectly predicted, only 0.021 away from a perfect score.
- GNP chunks are predicted very badly. The recall for GNP is only 9.4%, which means that only 9.4% of the total number of GNP chunks were correctly predicted. However, the precision of 40.2% show that nearly

Chunk type	$F_{\beta=1}$	Precision	Recall	Average length of correct chunks
NP	0.896	0.892	0.901	1.44
VP	0.979	0.977	0.980	1.38
PP	0.946	0.938	0.953	1.03
ADJP	0.842	0.826	0.859	1.22
ADVP	0.817	0.752	0.895	1.00
CONJP	0.907	0.889	0.926	1.00
INTERJP	0.948	0.913	0.992	1.00
SBUP	0.998	0.997	0.999	1.00
GNP	0.152	0.402	0.094	1.61
ALL	0.917	0.914	0.921	1.28

Table 5.6: Results at $S = 16000$

half of those that it did place, were placed correctly. The chunker also made 0.41 words longer chunks then the average in the NDT. Since a GNP "removes" part of a valid NP chunk, it might be interesting to try a chunker without the GNP chunks.

- The precision of both INTERJP and SBUP falls down from the baseline.
- Overall, the recall was slightly higher than the precision.
- Sometimes, the chunker places a chunk at a punctuation mark that clearly should be a O chunk. This lowers the precision, but does not affect the recall. This might be fixed if a rule were created so that punctuation marks should always be an O chunk.

5.4 Effect of different mutation and crossover rates on a single sentence

The higher the rate of mutation, a larger part of each offspring is mutated away from their parent. In the cases where there is a valley between the local optima, a high mutation rate has a higher chance to "leap across" these valleys, and hit the local optimum on the other side, which might have a higher fitness value. A high mutation rate, however, have a lower chance of doing small steps upwards, as the high rate mutates the offspring to be too far away. In that case, a low mutation rate might be better.

A high rate of crossover will create more offspring between individuals. A crossover might merge the two good parts from two parents, and create an offspring, which also can be mutated, that is better than either of those two parents. The difference between mutation rate and crossover is that a high mutation rate changes the offspring more than a low mutation rate, while a high crossover rate creates more offspring that are the product of two parents and at a low crossover rate, more offspring is created asexually, as clones of their single parent.

Here we show the effect of different mutation and crossover rates. To show this, we are using a single sentence, sentence 8215 in NDT version 0.3. This sentence is long and complex, and should therefore be a good indicator of the

effect of mutation and crossover on other long and complex sentences. It should also indicate the effect of crossover and mutation on all other sentences. We need to use a single sentence, as using more sentences would take too much time. It does nevertheless give a good representation of the effect of changing mutation and crossover rates. The stopping criteria is at $S = 250$, with a population size at 1678 and tournament size at 10.

Figure 5.1 shows how the F1 score of the result for sentence 8215 changes depending on both the crossover and mutation rate. The data were generated from an average of 60 runs with different seeds for each crossover rate and mutation rate, using sentence 8215. Doing these runs using the whole test-set would have taken too long time, around 100-200 days.

As figure 5.1 shows, the best results happen with a mutation rate of between 1% and 4%, with a crossover rate below 50%. With a crossover rate above 50%, the F1 score is worse. This is reversed when the mutation rate is at 6% or above, as there a higher crossover rate leads to better results. However, with a higher mutation rate than 4%, the results get progressively lower, and at a mutation rate of 12% and above, the F1 score is below 55%.

The low score at high mutation rates can be explained. The mutation rate is the chance that a single chunk-tag in the individual is mutated and, over many mutations, the mutation rate is the percentage of the individual that is mutated. Each mutation that happens, draws a new chunk randomly from the list of all possible chunks, a list consisting of 19 tags (see table 2.3). Nevertheless, for many word classes, only a small subset of all the possible chunk tags is actually possible for that word class. For example, a noun can only be either **B-NP** or **I-NP**. That means, if we have a noun and its chunk tag is mutated, the mutation only have **10.5%** chance of hitting an **B-NP** or **I-NP**, and all the other tags are not valid. Since the training set have no cases of a noun having anything other than **B-NP** or **I-NP**, any other chunk tags will give no fitness score for that word. In addition, since the formulas in 4.6.1 gives a single word more fitness if the words around that word have valid chunk tags, the mutation of a chunk tag into a tag not existing in the training set, will often also lower the fitness score of all the words around that word.

If we look at the fitness formulas f_4 and f_8 together, we see that they have a reach of seven words, three on each side of the word being mutated. This means that a mutation of a chunk-tag will often affect the fitness score from f_4 and f_8 for words up to three positions from the chunk. The rate of this happening depends on the mutation rate, but if we have a set of sentences with a uniform distribution of sentence lengths, a mutation rate of **10%** will on average change the fitness values of $\sim 40.0\%$[4] of the sentence, when using f_4 and f_8. A mutation rate of **6%** will change the fitness values of $\sim 25.8\%$ of the sentence, and a mutation rate of **4%** will change the fitness values of $\sim 18.1\%$ of the sentence. This results in that having a higher mutation rate than **4%** affects too much of the individual, giving the GA a low chance of climbing the slope towards a better fitness.

The increase in F1 score we get from raising the crossover rate when the mutation rate is more than **4%** is an effect of the way crossover works and the relatively low fitness score when a high mutation rate is in effect. When we

[4]These percentages were found using an empirical method. The average sentences length used were 17, with a standard deviation of 5.

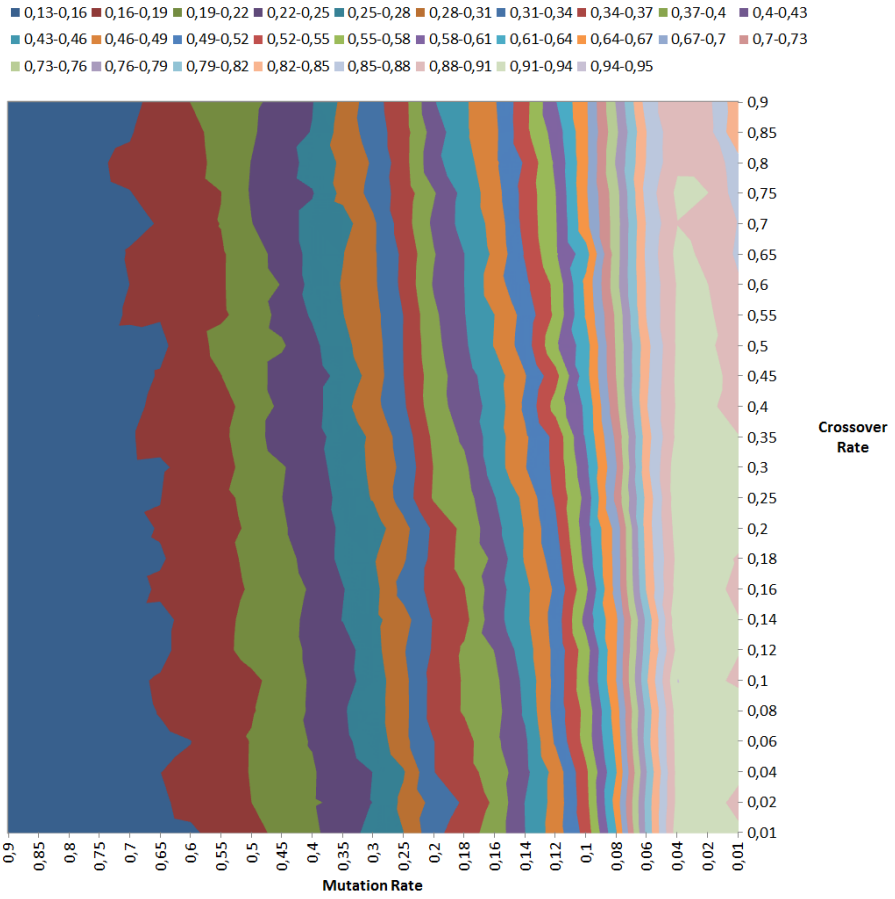


Figure 5.1: The resulting average F1 score from 60 runs of sentence 8215 with difference crossover- and mutation rates. The best resulting data point, at 4% mutation and 10% crossover, is a small gray dot.

have a high mutation rate, we have generally low fitness rates. This means that some parts of the individuals have been mutated correctly, while other parts of the individuals are not correct. Since crossover takes one part from the first parent and another part from the second, it has a chance to generate a new individual consisting of the good parts of the two parents, and having a better fitness score.

This effect is reversed when we have a good fitness score of both parents. As seen in figure 5.4, only a part of the population is at the best fitness value, when the fitness value is high. At that point, a very low number of mutations are left to do, to get a perfect[5] result, and the individuals in the population that have a lower fitness than the best do not contain these mutations.[6]Therefore, a crossover at that point will either create a member that is identical to the ones

[5]As good a result as can be had from the training set and the fitness functions

[6]The mutations would, by a high probability, already have been included in the best individuals

in the best part of the population, or create a member that is inferior. Since when crossover occurs, it reduces the number of individuals created asexually, in effect it reduces the total number of chances the mutation function have to get a perfect result. This lowers the final result, and creates the "dip" in the b-score seen in figure 5.1, at high crossover rates.

As the results above show, the mutation and crossover rates determined by the BVDGA are probably the correct ones.

5.5 Creating a better result

Here we will analyze the variance of the results we get from the chunker, and then use that analysis to create some methods to increase the final result. Specifically, we will see why some random choices give much better results than others do. The analysis and the methods presented were determined on the same dataset as in 5.2.1.1 and with the same parameters as in table 5.1 and table 5.2.

5.5.1 Analysis of the variance of the chunker

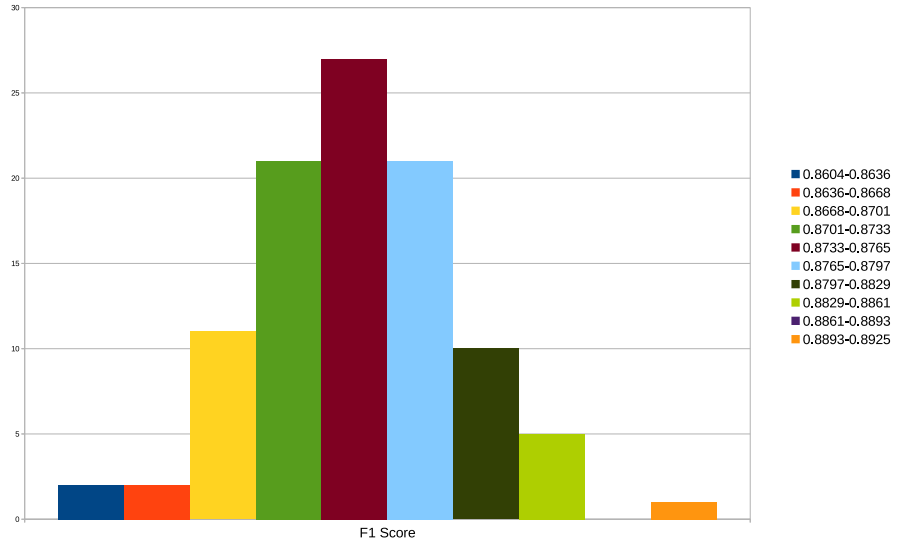
The chunker does many random choices, in random mutation, in crossover, and in the selection of parents. These choices determine how the chunker evolves the population, and the final output from the population varies depending on the different random choices the GA does. This section describes the different results that occur, depending on random fluctuations.

5.5.1.1 Variance of the results depending on different random choices

Here we are showing the results of different stopping criteria. See 4.8 for the stopping algorithm.

Variance of score at stop value at 250 The first result is from a very low stopping criteria, at $S = 250$. This low stopping value has a quite bad final result, and has a lot of variance between the different final productions. Each run was using a different seed value, giving different random choices. As figure 5.2 shows, at a stop value of $S = 250$, we get F1 scores with an amount of variance, with a standard deviation at 0.51%. Of note is also the best result, at **88.91%**, 1.73% higher than the average. As the result we got at 88.91% shows, our training set and evaluation method produces fitness values that can give at least 88.91%. Using different methods, we might be able to get all random choices give that good answer.

Variance and results at increasing stop criteria A simple way to increase the correctness of the result, and also lower the variance, is to increase the stop criteria. This also has the unfortunate result of greatly increasing the calculation time used. In figure 5.3 we have used a stop values at $S = 250$, $S = 2000$, $S = 16000$, $S = 128000$, $S = 1024000$. These values are simply the previous value times 2^3. The results levels out at $S = 1024000$ with only a small increase in F1 score between $S = 128000$ and $S = 1024000$, with an increase of 0.0491. Even higher stop values should only give insignificant increases in



Average	87.1777%
Standard deviation	0.51705%

Figure 5.2: Results from 100 runs of the GA, each with different random seeds, at a stop value of 250

the final result. We can note here that the variance between different random choices is much smaller at higher stopping criteria.

Stop Criteria	Average	Standard deviation
$S = 250$	87.1777%	0.51705%
$S = 2000$	88.5186%	0.45892%
$S = 16000$	90.8607%	0.34798%
$S = 128000$	92.1585%	0.15067%
$S = 1024000$	92.2076%	0.13988%

Table 5.7: Results from 5 different stop values, each with 100 different seeds, with stop values of 250, 2000, 16000, 128000, 1024000

5.5.1.2 Single sentence variance

To examine this variance further, we need to look at individual sentences in the test set, and see how they behave. For this, we use sentence number 8215 in the NDT version 0.3, the same as used in section 5.4. The F1 scores of this sentence are in table 5.8, and as shown the scores varies a lot. It also shows that, on sentence 8215, the training set can give a fitness score that results in a perfect result where all the chunk tags are correct. The difference between the perfect result, and the worst result are however large, at **22.22%**.

To understand the difference between the worst and perfect result we can look at the end state of the population, which is the fitness value of all the

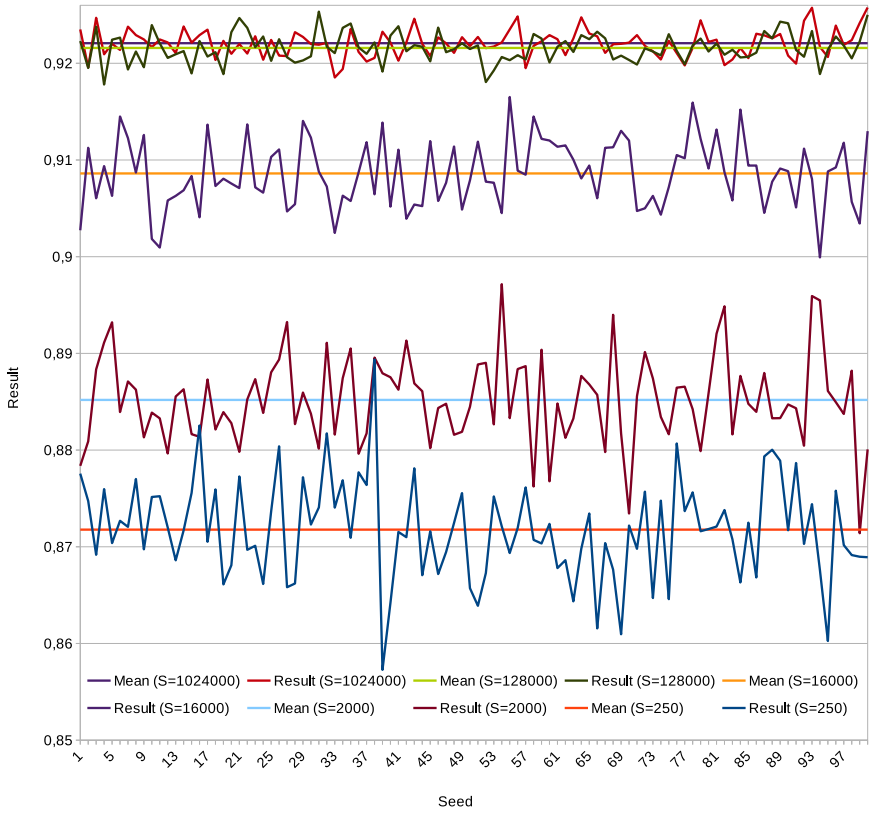


Figure 5.3: Results from 5 different stop values, each with 100 different seeds, with stop values of 250, 2000, 16000, 128000, 1024000

individuals in the population at the moment when we select the final result. In figure 5.4 we have the fitness values of all the members of the population at the moment when we select the final output from the chunker. Note here that we are using the raw fitness values produced by the chunker, and not the F1 score, which the chunker does not know. As explained in the formula for calculating the fitness of an individual in 4.6, a higher fitness value can give a lower F1 score, because the training set can create incorrect answers.

As shown in figure 5.4, the populations evolve up to plateaus, which are many individuals all with the same fitness values. These plateaus are local optima, which the GA needs to evolve away from to increase the fitness. As shown in the figure, the top plateau of the worst result is significantly lower than the top of the best result. The best result is at a fitness score of **4150**, while the worst is at **3438**, a difference of **712**. The randomness of mutation and the random initialization is the main cause of this. The worst result has either had a worse start, with a bad starting initialization, or has drawn "unluckily" in the random mutation, and is therefore at a lower plateau than the best result. The methods below will try to get the final population state to be more equal

Sentence 8215 standard deviation	0.0436
Sentence 8215 min F1 score	0.7878
Sentence 8215 max F1 score	1.0000

Table 5.8: $F_{\beta=1}$ score of sentence 8215. Stop value at 250.

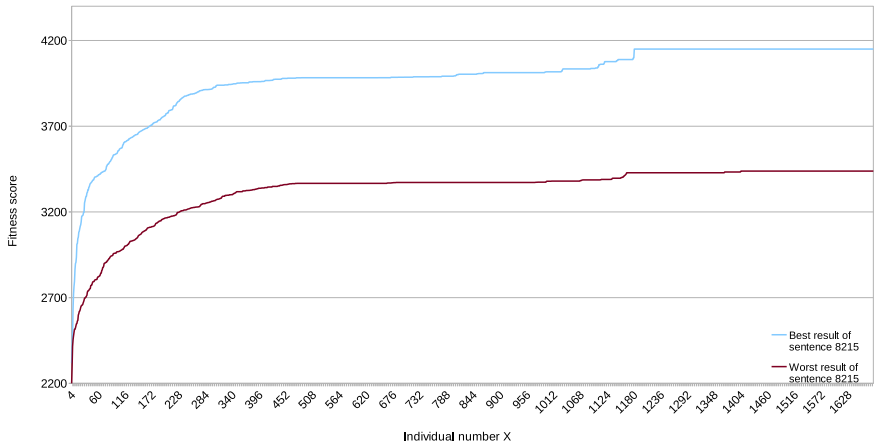


Figure 5.4: Final fitness values of all the members of the population at the end of the GA, ordered by fitness score

between different random choices.

5.5.2 Method for random dispersion of the population at high concentration.

As we saw in section 5.5.1.2, when the GA stops and produces a chunked sentence, different random seeds give very varying final populations, where one seed has a much higher final population fitness than the other. Here we will attempt a method to mitigate that effect.

The idea here is to, when x parts of the population are at identical fitness to the best performing individual, we will forcibly mutate that part of the population, and thereby spread the population outwards. This might give the GA more chance of climbing a different slope, and perhaps getting better result. This will of course cost more time, in the form of more generations used.

These experiments were done with a stopping criterion at $S = 250$, $S = 2000$ and $S = 16000$, and with the same set as used in 5.2.1.1. The results are in table 5.9, 5.10 and 5.11. To disperse the plateau, we are using the mutation operator detailed in 4.4, with a mutation chance at each gene at 50%.

The results increase significantly, with a significance level of < 0.01, from $x = 0\%$ to $x = 10\%$, but between $x = 10\%$ and $x = 50\%$ there is no statistical significant difference. The increase in generations used is only statistically significant between $x = 0\%$ and $x = 10\%$ at $S = 250$ and $S = 2000$, and the increase in generations used is very small, in the order of 1.5% increase.

These results show that this method can be used to give better results with a very low increase in the number of generations used.

x	$F_{\beta=1}$	Generations used
NA	0.8708	$9.84 * 10^6$
10%	0.8774	$9.95 * 10^6$
20%	0.8748	$9.92 * 10^6$
30%	0.8768	$9.90 * 10^6$
40%	0.8760	$9.96 * 10^6$
50%	0.8758	$9.88 * 10^6$

Table 5.9: Stop at $S = 250$. Showing results at different x. x is the percentage of the population that must be identical for the dispersion to occur. Generations used is the total number of generations used in that run.

x	$F_{\beta=1}$	Generations used
NA	0.8837	$1.99 * 10^7$
10%	0.8961	$2.03 * 10^7$
20%	0.8939	$2.01 * 10^7$
30%	0.8941	$2.02 * 10^7$
40%	0.8962	$2.03 * 10^7$
50%	0.8948	$2.01 * 10^7$

Table 5.10: Stop at $S = 2000$. Showing results at different x. x is the percentage of the population that must be identical for the dispersion to occur

x	$F_{\beta=1}$	Generations used
NA	0.9085	$1.05 * 10^8$
10%	0.9159	$1.05 * 10^8$
20%	0.9154	$1.03 * 10^8$
30%	0.9141	$1.04 * 10^8$
40%	0.9143	$1.04 * 10^8$
50%	0.8948	$1.03 * 10^8$

Table 5.11: Stop at $S = 16000$. Showing results at different x. x is the percentage of the population that must be identical for the dispersion to occur

5.5.3 Multiple runs

As we can see in 5.5.1.2, a sentence can show a large amount of variance between different runs, depending on the random choices done. A really simple way to get better results at each sentence is then to run the chunker several times at each sentence with different seeds, then selecting the best individual of those runs as the output from the chunker. This attacks the discrepancy we see in figure 5.4, by running the sentence through the chunker several times, and selecting the best result. This method is really simple to implement, but it of course increases the number of generations used greatly.

The results are in tables 5.12, 5.13 and 5.14. Significance testing was done between each value of *Runs*, at a confidence level of $> 95\%$. The results stabilized at $Runs = 12$ with $S = 250$, at $Runs = 16$ with $S = 2000$, and at $Runs = 8$ at $S = 16000$, with further increases in *Runs* not being statistical significant. Note that the number of generations used increases linearly with the increase in *Runs*.

Runs	$F_{\beta=1}$ (increase from $Runs = 1$)	Generations used (% increase from $Runs = 1$)
1	0.8721 (.00000)	$9.83 * 10^6$ (0%)
2	0.8973 (.02511)	$1.96 * 10^7$ (99%)
3	0.9081 (.03591)	$2.97 * 10^7$ (201%)
4	0.9133 (.04118)	$3.95 * 10^7$ (301%)
5	0.9165 (.04436)	$4.94 * 10^7$ (402%)
6	0.9186 (.04642)	$5.92 * 10^7$ (501%)
8	0.9209 (.04875)	$7.90 * 10^7$ (703%)
12	0.9240 (.05189)	$1.18 * 10^8$ (1104%)
16	0.9244 (.05228)	$1.58 * 10^8$ (1507%)
20	0.9256 (.05348)	$1.97 * 10^8$ (1905%)

Table 5.12: Stop at $S = 250$

Runs	$F_{\beta=1}$ (increase from $Runs = 1$)	Generations used (% increase from $Runs = 1$)
1	0.8846 (.00000)	$1.99 * 10^7$ (0%)
2	0.9075 (.02256)	$3.97 * 10^7$ (99%)
3	0.9140 (.02905)	$5.96 * 10^7$ (199%)
4	0.9192 (.03424)	$7.94 * 10^7$ (298%)
5	0.9207 (.03574)	$9.95 * 10^7$ (399%)
6	0.9232 (.03831)	$1.19 * 10^8$ (499%)
8	0.9245 (.03953)	$1.59 * 10^8$ (698%)
12	0.9255 (.04058)	$2.39 * 10^8$ (1098%)
16	0.9268 (.04184)	$3.18 * 10^8$ (1495%)
20	0.9266 (.04168)	$3.98 * 10^8$ (1900%)

Table 5.13: Stop at $S = 2000$

5.5.3.1 Score compared to generations used

In figure 5.5, and with the tables 5.12-5.14, we have the relationship between F1 scores and the number of generations used to generate them. We can see that $S = 250$ get almost as good results as $S = 2000$ at a high number of runs, but uses only half as many generations to do so. We can also see that $S = 16000$ do not give a significantly better result, but $S = 16000$ use much longer time to do so. In essence, we can get as good results with more runs as with a high stopping value, but more runs will used far less time.

Runs	$F_{\beta=1}$ (increase from $Runs = 1$)	Generations used (% increase from $Runs = 1$)
1	0.9090 (.00000)	$1.06 * 10^8$ (0%)
2	0.9212 (.01213)	$2.11 * 10^8$ (99%)
3	0.9250 (.01592)	$3.16 * 10^8$ (199%)
4	0.9261 (.01708)	$4.24 * 10^8$ (301%)
5	0.9267 (.01768)	$5.31 * 10^8$ (402%)
6	0.9265 (.01749)	$6.35 * 10^8$ (500%)
8	0.9271 (.01801)	$8.52 * 10^8$ (706%)
12	0.9273 (.01826)	$1.28 * 10^9$ (1107%)
16	0.9271 (.01808)	$1.70 * 10^9$ (1510%)

Table 5.14: Stop at $S = 16000$

5.5.4 Different fitness formula

The fitness formula described in 4.6.1 do not weight the different features drawn. A fitness formula that includes weights might do better.

5.5.4.1 The formula

The formula here is an expansion of the one described in 4.6.1, and the difference is in the function $f_i(x)$:

$$f_i(x) = \begin{cases} w_i(ln(m_i) + ln(c_i(x))) & if \quad c_i(x) > 0 \quad and \quad m_i > 0 \\ e_i & if \quad c_i(x) = 0 \quad and \quad m_i > 0 \\ 0 & if \quad c_i(x) = \text{undefinded} \quad or \quad m_i \leq 0 \end{cases} \tag{5.1}$$

w_i is a value weighting each function. m_i is a number that is used to smooth each function. e_i is a value used when $occ_i(x) = 0$. $c_i(x)$ is as described in 4.6.1. $c_i(x) = \emptyset$ triggers when $c_i(x)$ would go outside the bounds of the sentence.

The rest of the formula is as described in 4.6.1.

5.5.4.2 Determining the parameters

The parameters in formula 5.1 had to be determined. To do so, the BVDGA from 5.2.1.2 was used, with these modifications:

- The values for mutation rate, crossover rate, population size and tournament size were as in 5.2
- The parameters were initialized to a value of 10, then immediately mutated using the same method as in the mutation part of 5.2.1.2. A value of 100 was used as the standard deviation in the mutation.
- The mutation operator was the same as in 5.2.1.2, using a value of $\frac{1}{15}$ of the current value as the standard deviation. A mutation chance of 100% was used.
- The crossover was the same as in 5.2.1.2, at 40% chance.

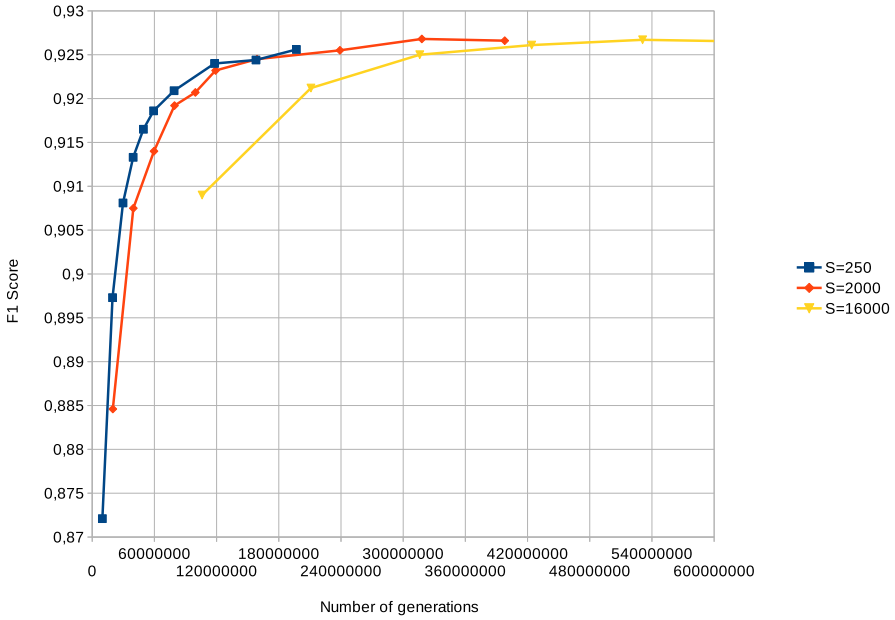


Figure 5.5: F1 Score at different stop values compared to generations used. The upper scores from $S = 16000$ is not shown.

- A stop value of $S = 2000$ was used, and the method in 5.5.3 was used, with the number of runs at $Runs = 8$.

The determined parameters are in table 5.15

i	1	2	3	4	5	6	7	8	9	10	11
m_i	13	62	15	2	31	208	53	0.0001	21	9	346
w_i	54	5	88	93	8	133	55	-31	29	26	751
e_i	45	96	24	69	15	-51	-92	78	83	97	45

Table 5.15: Parameters of extended fitness formula

5.5.4.3 Results

The results are in table 5.16. If we compare them to the results presented in table 5.9, 5.10 and 5.11 at $x = 0\%$, we can see that the increase is in the range of 0.30-0.70. If we compare the parameters with the parameters in table 5.1, we can see that none of the m_i parameters is 0 or below. This means that, compared to the parameters in 5.1, none of the $c_i(x)$ functions returns 0. This means that the fitness generator will need to look up more data, which again takes more time. In effect, this extended fitness formula takes at least twice as long as the normal one per run of the fitness formula. The exact time used depends on the machine it is running on.

Stop criteria	$F_{\beta=1}$	Generations used
250	0.8843	$9.63 * 10^6$
2000	0.8906	$1.92 * 10^7$
16000	0.9125	$1.04 * 10^8$

Table 5.16: Scores using the extended fitness formula.

5.6 Better results

Here is the result from using the methods from 5.5. The same setup was used as in 5.3, with the same dataset. The best parameters that cost the least in terms of generations used from each of the methods were chosen. Since $S = 16000$ only gave insignificant increases, $S = 16000$ was not used. Note that at $Runs = 12$ and $Runs = 16$, the results was much more stable, so that fewer runs where needed to get statistical significance.

Method used	$F_{\beta=1}$	Generations used
No methods used	0.875	$4.99 * 10^7$
Dispersion, $x = 10\%$	0.884	$5.09 * 10^7$
Multiple runs, $Runs = 12$	0.937	$5.99 * 10^8$
New fitness formula	0.888	$4.80 * 10^7$
$x = 10\%$, $Runs = 12$, new fitness formula	0.937	$5.85 * 10^8$

Table 5.17: Results from using methods to increase the accuracy, at $S = 250$

Method used	$F_{\beta=1}$	Generations used
No methods used	0.887	$1.16 * 10^8$
Dispersion, $x = 10\%$	0.901	$1.18 * 10^8$
Multiple runs, $Runs = 16$	0.938	$1.86 * 10^9$
New fitness formula	0.895	$1.13 * 10^8$
$x = 10\%$, $Runs = 16$, new fitness formula	0.937	$1.82 * 10^9$

Table 5.18: Results from using methods to increase the accuracy, at $S = 2000$

As we can see from the results in tables 5.17 and 5.18, the best score is at $Runs = 16$ with $S = 2000$. This score is however very close to three other scores, as we can see in table 5.19.

Since the scores in table 5.19 are nearly identical, even with many different parameters, we can conclude the $F_{\beta=1}$ score at 0.938 is the maximum achievable with the features used by the fitness functions, namely only the Part of Speech of the words. To increase the score more, a fitness formula that uses more features will have to be created instead. These features can for example be the words themselves, or extra lexical information.

In table 5.20 is the detailed results from each chunk class using one of the methods in table 5.19. Since the methods in that table give nearly identical results, the one with the least time spent was used. That is at $S = 250$ and $Runs = 12$. This is because the extended fitness formula from 5.5.4 spends at least twice as much time per fitness calculation as the more simple fitness

Method used	$F_{\beta=1}$	Generations used
$S = 250$, $Runs = 12$	0.937	$5.99 * 10^8$
$S = 250$,$x = 10\%$, $Runs = 12$, new fitness f.	0.937	$5.85 * 10^8$
$S = 2000$, $Runs = 16$	0.938	$1.86 * 10^9$
$S = 2000$, $x = 10\%$, $Runs = 16$, new fitness f.	0.937	$1.82 * 10^9$

Table 5.19: Nearly identical results

formula.

Note the low scores for the GNP chunk type. While it is not that many GNP chunks in the dataset, a slight improvement in score could be achieved by converting those chunks into standard NP chunks.

Chunk type	$F_{\beta=1}$	Precision	Recall
NP	0.919	0.916	0.922
VP	0.989	0.989	0.989
PP	0.952	0.953	0.962
ADJP	0.901	0.908	0.893
ADVP	0.858	0.804	0.919
CONJP	0.940	0.930	0.951
INTERJP	0.963	0.941	0.990
SBUP	0.999	0.998	1.000
GNP	0.164	0.539	0.097
ALL	0.937	0.936	0.937

Table 5.20: Results using $S = 250$ and $Runs = 12$

5.7 Effect of removing the GNP chunks

As we can see from the results in table 5.20, the chunker are not able to predict the GNP chunks, managing only to recall 9.7% of the GNP chunks. To see the effect the existence of GNP chunks have on the result, the GNP chunk can be removed.

```
NP[Thorvald Hansens] GNP[grønne bil]
```

Is converted into

```
NP[Thorvald Hansens grønne bil]
```

This removes the GNP chunks, and makes a longer NP chunk. The effect this has on the results are in table 5.21. As can be seen, the results improve with about 1% point, with the NP chunks giving all of that improvement. The conclusion is that the fitness formula used by the chunk do not extract enough information to be able to chunk the GNP effectively.

Chunk type	$F_{\beta=1}$	Precision	Recall
NP	0.938	0.936	0.935
VP	0.988	0.988	0.988
PP	0.953	0.945	0.962
ADJP	0.897	0.908	0.887
ADVP	0.857	0.804	0.916
CONJP	0.946	0.945	0.948
INTERJP	0.995	0.992	0.999
SBUP	0.999	0.999	1.000
GNP	NA	NA	NA
ALL	0.950	0.946	0.954

Table 5.21: Results without GNP, using $S = 250$, $Runs = 12$ and the new fitness formula

Chapter 6

Chunking part of the Penn Treebank

6.1 Introduction

The chunking of the NDT in chapter 5 gave a $F_{\beta=1}$ score of max 0.938 using the best chunker. However, this score is not compared to any other methods. To compare the GA chunker to other chunkers, the GA chunker chunked the same set as was used in the ConLL2000 task.

6.2 Chunking the ConLL2000 set using the same parameters as were discovered in chapter 5

The data was chunked using the same parameters as detailed in 5.1 and 5.2. The method for calculating the accuracy and precision of the result is the same as described in 5.3. The baseline is made by choosing the most common chunk tag as that word's tag.

Chunk type	$F_{\beta=1}$	Precision	Recall	Average length of correct chunks
NP	0.832	0.799	0.868	1.988
VP	0.667	0.605	0.742	1.231
PP	0.844	0.747	0.971	1.000
ADVP	0.565	0.443	0.777	1.000
ADJP	N/A	N/A	N/A	N/A
CONJP	N/A	N/A	N/A	N/A
SBAR	N/A	N/A	N/A	N/A
INTJ	0.500	0.500	0.500	1.000
LST	N/A	N/A	N/A	N/A
PRT	0.153	0.750	0.085	1.000
ALL	0.771	0.726	0.821	1.585

Table 6.1: Baseline for ConLL2000 set

Chunk type	$F_{\beta=1}$	Precision	Recall	Average length of correct chunks
$S = 250$	0.809	0.799	0.820	1.668
$S = 2000$	0.820	0.811	0.830	1.669
$S = 16000$	0.851	0.843	0.858	1.687

Table 6.2: Results at different stop values for ConLL2000 set

The baseline in table 6.1 is significantly better than the baseline for the NDT. This baseline also manages to predict some chunks longer than 1.0. But several chunk classes are not chunked at all in the baseline.

In table 6.2 is the results from stop values at $S = 250$, $S = 2000$ and $S = 16000$. None of them gives a "good" result when compared to the results in the ConLL2000 task. The results are therefore not analyzed further, but we note that chunking using the same parameters as in the Norwegian set do not give good results.

6.3 Chunking the ConLL2000 with new parameters

Using the same parameters as in the Norwegian set is, as seen above, not ideal. A new set of parameters is therefore needed to chunk the ConLL2000 dataset.

6.3.1 Determining the parameters

To determine the parameters for chunking the ConLL2000 dataset, the same method as described in 5.2.1 was used. But instead of determining the parameters to the formula from 4.6.1, the parameters to the extended formula from 5.5.4 was determined. The differences from the method used in 5.2.1 is detailed below. Note that only the parameters to the fitness formula were determined, the GA parameters were kept from the chunking of the NDT.

Dataset used

The dataset used by the ConLL2000 task consists of a training set of 8936 sentences, and a test set of 2012 sentences. To generate the parameters, 100 sentences were randomly chosen from the training set, while the rest of the training set was used to calculate fitness. The test set was not used to generate parameters.

Initialization

The parameters were initialized to a value chosen from a normal distribution with a mean of 10 and a standard deviation of 1000. If any of the m_i parameters were below 0, it were set to 0.0001. Note that this sets quite many of the m_i parameters to 0.0001.

Mutation

The mutation operator was the same as in section 5.2.1.2., but using a mutation standard deviation of $\frac{1}{10}$ of the current value.

Crossover

Identical as the one in 5.2.1.2.. Crossover chance at 40%

Parent selection and population model

This used the same population model as the population model in 5.2.1.2.

Fitness calculation

For fitness calculation, the methods used to improve the results from section 5.5 was used. To calculate the fitness, the chunker was run with the following parameters:

- $Mutationrate = 0.04$
- $Crossoverrate = 0.001$
- $Populationsize = 1678$
- $Tournamentsize = 10$
- $Runs = 8$
- $Stopcrit = 4000$
- Break apart with $x = 10\%$
- Extended formula from 5.5.4, using the individual's parameters

The resulting $F_{\beta=1}$ score was the fitness of the individual.

6.3.1.1 Resulting parameters

The parameters in table 6.3 were discovered. Because of time constraints, these parameters were determined in under a week, far less than the parameters used to chunk the NDT. This will affect the result.

i	1	2	3	4	5	6	7	8	9	10	11
m_i	32	1665	979	302	0.00012	0.00010	990	0.00017	303	13	11
w_i	198	487	1834	-939	-1145	228	342	-1525	321	20	1196
e_i	-924	1316	1876	244	-440	-447	-228	-327	989	9	434

Table 6.3: Parameters of extended fitness formula

6.3.2 Results from chunking the ConLL2000 set

The results in table 6.4 is from the test set in the ConLL2000 task. To get as good results as possible, all the methods from section 5.6 were used, with $x = 10\%$, $Runs = 16$, the parameters in table 6.3, and a stop value at $S = 4000$.

The results are somewhat lower than most of the results in the ConLL2000 task, ranked 3rd last. In table 6.5 are the closest result upward in the ranking. As can be seen, the GA chunker chunks NP slightly better than that chunker, but the rest somewhat worse. This can be attributed to that the GA chunker used only the POS tags, and nothing else, while the chunkers used in the ConLL2000 task used several other features. A better GA could be created using more features than just the Part of Speech tags.

Chunk type	$F_{\beta=1}$	Precision	Recall	Average length of correct chunks
NP	0.910	0.907	0.913	2.117
VP	0.910	0.901	0.919	1.540
PP	0.912	0.883	0.944	1.000
ADVP	0.723	0.684	0.766	1.004
ADJP	0.622	0.674	0.578	1.245
CONJP	0.109	0.300	0.067	1.709
SBAR	0.445	0.594	0.356	1.000
INTJ	0.387	0.317	0.500	1.000
LST	N/A	N/A	N/A	N/A
PRT	0.256	0.450	0.179	1.000
ALL	0.888	0.883	0.892	1.709

Table 6.4: Results for ConLL2000 set. LST chunks do not exist in the test set, and is therefore not counted.

Chunk type	$F_{\beta=1}$	Precision	Recall
NP	0.898	0.902	0.894
VP	0.916	0.915	0.916
PP	0.955	0.959	0.951
ADVP	0.768	0.797	0.768
ADJP	0.698	0.729	0.669
CONJP	0.500	0.400	0.667
SBAR	0.795	0.821	0.770
INTJ	1.00	1.000	1.000
LST	N/A	N/A	N/A
PRT	0.667	0.821	0.770
ALL	0.901	0.906	0.897

Table 6.5: Pla and Molina's results from ConLL2000 chunking task

Conclusion

This thesis has been divided into two parts. In part I, we have created a chunk structure on the Norwegian Dependency Treebank. To do so a set of deterministic rules where created, which was then applied to the NDT, creating a chunk structure on the text in the NDT. The created chunk structure should be correct and other can use this structure in their own work. The rules we created should be verified by others however.

Part II is an attempt at creating a chunker, using a Genetic Algorithm. This chunker was developed and tested on the output from part I, the chunk structure on the text in the NDT. Two different fitness formulas and several different parameters to the GA were tested, as were two methods of improving the GA. Amongst the methods tested, running the GA several times on the same sentence gave good improvement of the result, but this came at a great cost. The method for breaking apart a static population was also tested; this method gave a slight improvement with only a slight cost. Overall, the chunker performed quite well on the output from part I, correctly predicting about 93% of all chunks. Unfortunately, no other chunking methods have been tested. The GA chunker has therefore not been tested against other chunking methods.

We also tested the GA against the English text used in the ConLL2000 task. The same methods were used as on the NDT, and the same fitness method. New fitness parameters were however determined. The results were, however, not so good, coming 3rd last in the list.

Overall, a GA can be said to be a viable method for chunking. However, the GA operators, the parameters of the GA, and especially the formula for calculation fitness, needs to be carefully constructed to give a good result. A GA can also be computationally intensive, depending on how the fitness function is constructed.

Bibliography

Abney, Steven P. (1991). "Parsing by Chunks". In: *Priciple-Based Parsing*. Ed. by Robert C. Berwick, Steven P. Abney, and Carol Tenny. Dordrecht, Netherlands: Kluwer Academic Publishers, pp. 257–268.

— (1996). *Chunk Stylebook*. (accessed May 28, 2013). URL: `http://www.vinartus.net/spa/96i.pdf`.

Araujo, Lourdes (2002a). "A Parallel Evolutionary Algorithm for Stochastic Natural Language Parsing". In: *Parallel Problem Solving from Nature — PPSN VII*. Ed. by JuanJuliánMerelo Guervós et al. Vol. 2439. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 700–709. ISBN: 978-3-540-44139-7. DOI: `10.1007/3-540-45712-7_67`. URL: `http://dx.doi.org/10.1007/3-540-45712-7_67`.

— (2002b). "Part-of-Speech Tagging with Evolutionary Algorithms". In: *Computational Linguistics and Intelligent Text Processing*. Ed. by Alexander Gelbukh. Vol. 2276. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 230–239. ISBN: 978-3-540-43219-7. DOI: `10.1007/3-540-45715-1_21`. URL: `http://dx.doi.org/10.1007/3-540-45715-1_21`.

Bird, Steven, Martin Klein, and Edward Loper (2009). *Natural Language Processing with Python*. O'Reilly Media.

Brill, Eric (1993). "A Corpus-Based Approach To Language Learning". PhD thesis. University of Pennsylvania.

— (1994). "Some advances in rule-based part of speech tagging". In: *Proceeding of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*. Seattle, Washington, USA.

Church, Kenneth (1988). "A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text". In: *Second Conference on Applied Natural Language Processing*. Austin, Texas, USA.

Déjean, Hervé (2000a). "ALLiS: a Symbolic Learning System for Natural Language Learning". In: *Proceedings of CoNLL-2000 and LLL-2000*. Lisbon, Portugal.

— (2000b). "Learning Syntactic Structures with XML". In: *Proceedings of CoNLL-2000 and LLL-2000*.

Eiben, A. E. and J.E. Smith (2007). *Introduction to Evolutionary Computing*. Springer.

Ferran Pla, Antonio Molina and Natividad Prieto (2000). "Improving Chunking by Means of Lexical-Contextual Information in Statistical Language Models". In: *Proceedings of CoNLL-2000 and LLL-2000*.

Fortin, Félix-Antoine et al. (2012). "DEAP: Evolutionary Algorithms Made Easy". In: *Journal of Machine Learning Research* 13, pp. 2171–2175.

GuoDong Zhou, Jian Su and TongGuan Tey (2000). "Hybrid Text Chunking". In: *Proceedings of CoNLL-2000 and LLL-2000*.

Halteren, Hans van (2000). "Chunking with WPDV Models". In: *Proceedings of CoNLL-2000 and LLL-2000*.

Haug, D. T. T. (2010). *PROIEL Guidelines for Annotation*. URL: `http://folk.uio.no/daghaug/syntactic_guidelines.pdf`.

Johansson, Christer (2000). "A Context Sensitive Maximum Likelihood Approach to Chunking". In: *Proceedings of CoNLL-2000 and LLL-2000*.

Jurafsky, Daniel and James Martin (2009). *Speech and Language Processing*. Upper Saddle River, New Jersey, USA: Person Education, Inc.

Knapsack Problem. URL: `http://en.wikipedia.org/wiki/Knapsack_problem`.

Koeling, Rob (2000). "Chunking with Maximum Entropy Models". In: *Proceedings of CoNLL-2000 and LLL-2000*.

Kudoh, Taku and Yuji Matsumoto (2000). "Use of Support Vector Learning for Chunk Identification". In: *Proceedings of CoNLL-2000 and LLL-2000*.

Norsk dependenstrebank (2014). Nasjonalbiblioteket.

Osborne, Miles (2000). "Shallow Parsing as Part-of-Speech Tagging". In: *Proceedings of CoNLL-2000 and LLL-2000*.

Ramshaw, Lance A. and Mitchell P. Marcus (1995). "Text chunking using transformation-based learning". In: *Third ACL Workshop on Very Large Corpora*. Cambridge, Massachusetts, USA.

Sang, Erik F. Tjong Kim (2000). "Text Chunking by System Combination". In: *Proceedings of CoNLL-2000 and LLL-2000*.

Sang, Erik F. Tjong Kim and Sabine Buchholz (2000). "Introduction to the CoNLL-2000 Shared Task: Chunking". In: *Proceedings of CoNLL-2000 and LLL-2000*. Lisbon, Portugal.

Traveling Salesman Problem. URL: `http://en.wikipedia.org/wiki/Travelling_salesman_problem`.

Veenstra, Jorn and Antal van den Bosch (2000). "Single-Classifier Memory-Based Phrase Chunking". In: *Proceedings of CoNLL-2000 and LLL-2000*.

Vilain, Marc and David Day (2000). "Phrase Parsing with Rule Sequence Processors: an Application to the Shared CoNLL Task". In: *Proceedings of CoNLL-2000 and LLL-2000*.